UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Ingeniería del Software e Inteligencia Artificial

TESIS DOCTORAL

Herramientas educativas para facilitar la adopción de la ingeniería de lenguajes software entre los desarrolladores informáticos

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Daniel Rodríguez Cerezo

Director

José Luis Sierra Rodríguez

Madrid, 2017

UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

Departamento de Ingeniería del Software e Inteligencia Artificial

HERRAMIENTAS EDUCATIVAS PARA FACILITAR LA ADOPCIÓN DE LA INGENIERÍA DE LENGUAJES SOFTWARE ENTRE LOS DESARROLLADORES INFORMÁTICOS

TESIS DOCTORAL

Presentada por:
Daniel Rodríguez Cerezo

Bajo la dirección del Doctor:
José Luis Sierra Rodríguez

Madrid, 2015

HERRAMIENTAS EDUCATIVAS PARA FACILITAR LA ADOPCIÓN DE LA INGENIERÍA DE LENGUAJES SOFTWARE ENTRE LOS DESARROLLADORES INFORMÁTICOS

Memoria que presenta para optar al título de Doctor en Informática:
Daniel Rodríguez Cerezo

Bajo la dirección del Doctor:
José Luis Sierra Rodríguez

Universidad Complutense de Madrid
Facultad de Informática
Departamento de Ingeniería del Software e Inteligencia Artificial

Madrid, 2015

A mi familia y mis amigos

Agradecimientos

Tras todo el tiempo y esfuerzo invertido en este trabajo de tesis, creo que es importante dar las gracias a toda la gente que siempre me ha apoyado y animado a lograrlo.

En primer lugar me gustaría dar las gracias a José Luis Sierra, no sólo por ser un buen mentor y amigo. También me gustaría agradecer su dedicación y paciencia, que sin duda han impulsado mi trabajo hasta llegar a esta tesis. También quiero agradecer su amistad y los grandes momentos y risas que hemos compartido trabajando juntos. Y por último, quiero agradecer la confianza que ha depositado en mí durante estos años, en los cuales he aprendido de él todo sobre lo que es investigar y trabajar en la Universidad. Estoy muy agradecido de haber podido contar con su apoyo y poder trabajar con él. Muchas gracias José Luis.

En segundo lugar quiero agradecer a todas las personas que han compartido estos años de trabajo de tesis, con sus idas y venidas. Todas estas personas me han ayudado de una u otra forma a hacer más llevaderos estos años de esfuerzo y trabajo. Aunque no puedo nombrar a todos de forma explícita, no quiero dejar de mencionar a algunos de ellos.

Me gustaría empezar por nombrar a mis compañeros del grupo de investigación ILSA. En especial a Antonio Sarasa, que ha colaborado en algunos de los resultados presentados en esta tesis y que siempre que lo he necesitado me ha ofrecido su ayuda. También a Joaquín y Bryan, con los que he compartido experiencias como doctorandos dentro del grupo.

A mis compañeros de la Universidad Complutense de Madrid, con los que he trabajado codo con codo. En particular a Rubén Fuentes y Ana Fernández-Pampillón. Y una mención especial para Mercedes Gómez Albarrán, que ha colaborado en la consecución de algunos resultados de esta tesis.

A mis compañeros de despacho 420VIP: Susana Bautista, Carlos Cervigon, Miguel Ballesteros, Iván Martínez y Carlos León. Desde el primer día que entré en el despacho hicieron que me sintiera como uno más y con ellos he compartido de todo en estos cinco años. Muchas gracias compis.

A Ana Bartolomé, Carlos Roa, Javier Muñoz, Luis Hernández e Iñaki, con los que he compartido risas a la hora previa a la comida.

A mis compañeros de carrera, que siempre han estado al tanto de mi progreso y con los que he compartido grandes momentos. Se merecen una mención especial: Ángel Valero y Rafael Fernández, por participar en el germen inicial de parte de este trabajo de tesis; Jorge Olmos, que siempre me ha apoyado durante estos años; y Daniel Garijo y Guillermo Frontera, que también se embarcaron en el doctorado.

A mis compañeros de baile: Abraham, David, Mar, Gema, Laura O., Laura G., Toni, Juan Antonio, Alicia, Javi, Belén, Juan y Cris. A ellos les agradezco que compartan esta pasión conmigo y que siempre estuviesen al tanto para ayudarme a desconectar de vez en cuando.

A mi familia, que siempre ha estado ahí apoyándome en los buenos y en los malos momentos. En especial a mi abuela Lola, a la que tengo un cariño especial, y

a mi prima Lidia, que seguramente me dejará “muy guapete” para la lectura de este trabajo.

Finalmente, y puesto de honor, para mis amigos de siempre. Tanto los que lleváis ahí toda la vida, como los que se han ido incorporando con el paso del tiempo. Miguel Ángel, Alex, Tortosa, Alfonso, Mario y Alicia; sin vosotros este camino, y los que tengan que venir, no serían lo mismo. Para mí es un orgullo el haberos conocido y agradezco enormemente todo el tiempo que pasamos juntos y vuestro apoyo casi incondicional.

Daniel Rodríguez Cerezo

Madrid, 24 de septiembre de 2015

Resumen

Históricamente, la materia de construcción de compiladores y procesadores de lenguaje es considerada por los estudiantes de Ingeniería en Informática como una materia difícil. Esto es debido, por una parte, a la naturaleza formal de las herramientas de especificación y diseño utilizadas, y, por otra, a la aplicación adecuada de diferentes técnicas sistemáticas de desarrollo para obtener los procesadores finales a partir de sus especificaciones. En esta tesis se aborda esta problemática en el caso particular de las gramáticas de atributos como formalismo básico de especificación.

La tesis aborda, primeramente la concepción de una estrategia para facilitar la comprensión de los aspectos básicos de las especificaciones basadas en gramáticas de atributos, así como el soporte software de dicha estrategia. La estrategia propone un enfoque dirigido por problemas, en los que el alumno debe emular el proceso de evaluación semántica sobre distintos supuestos de procesamiento de frases de acuerdo con gramáticas de atributos. Para soportar dicha estrategia, se ha desarrollado un sistema denominado *Evaluators*, que, tomando como entrada baterías de ejercicios sobre evaluación semántica, produce automáticamente simuladores interactivos que los estudiantes pueden utilizar para resolver dichos ejercicios. El sistema proporciona, así mismo, una herramienta de autoría, que puede ser utilizada por los docentes para proporcionar los ejercicios, así como una herramienta de análisis, que permite trazar el comportamiento de los estudiantes durante la resolución de los mismos en los simuladores generados. Respecto a dichos simuladores, la herramienta es capaz de generar simulaciones de dos tipos: simuladores basados en juegos serios, y simulaciones interactivas basadas en las representaciones abstractas convencionales utilizadas en la materia. La tesis presenta, así mismo, diversos resultados de evaluación de la herramienta, tanto con estudiantes como con docentes, que evidencian la utilidad práctica de la misma. Por último, la tesis abstrae también el modelo de proceso utilizado en la construcción de la misma, para facilitar la extrapolación del enfoque para soportar otros formalismos de especificación, o incluso a otros dominios diferentes del de los Procesadores de Lenguaje.

La tesis aborda, así mismo, los aspectos relativos a la creación de las especificaciones. Para tal fin, introduce un enfoque de desarrollo de procesadores de lenguaje dirigido por especificaciones basadas en gramáticas de atributos, que propugna la conservación de la especificación durante el proceso de desarrollo, así como el desarrollo basado en la evolución de prototipos. Para ello, el enfoque propone un patrón directo de codificación de gramáticas de atributos no circulares arbitrarias como esquemas de traducción, así como la generación de procesadores operativos a partir de dichos esquemas de traducción, mediante herramientas convencionales de generación de traductores (p.e., JavaCC o CUP). Las codificaciones resultantes mantienen, por una parte, una relación directa con las gramáticas de atributos originales, y, por otra, permiten a los alumnos experimentar directamente con prototipos ejecutables de sus especificaciones, así como hacer evolucionar dichos prototipos mediante la realización de diversas optimizaciones. Dichas codificaciones se ven facilitadas por el uso de una biblioteca denominada *EvLib*, que facilita el acoplamiento de un proceso de

evaluación semántica durante el análisis sintáctico llevado a cabo por los modelos de análisis sintáctico más convencionales (tanto descendentes como ascendentes). La técnica ha sido evaluada también positivamente tanto con estudiantes como con docentes en la materia.

Por último, la tesis aúna las dos contribuciones anteriores mediante la creación de un entorno educativo para el desarrollo de procesadores de lenguaje a partir de sus especificaciones basadas en gramáticas de atributos. Dicho entorno, que se ha denominado *EvDebugger*, permite generar automáticamente esquemas de traducción que implementan prototipos operativos de procesadores de lenguaje a partir de sus especificaciones dadas como gramáticas de atributos. Dicho proceso de generación utiliza los patrones de codificación de gramáticas de atributos como esquemas de traducción a los que se ha aludido anteriormente, facilitados mediante la biblioteca *EvLib*. El entorno permite también experimentar con el proceso de evaluación semántica, adoptando la estrategia de comprensión subyacente a *Evaluators*. Las evaluaciones preliminares de la herramienta con alumnos de la materia evidencian, así mismo, la utilidad práctica de la misma.

Abstract

Historically, the subjects related with the construction of compilers and language processors are considered as hard concepts to assimilate by students of Computer Science Engineering. This is due, on one hand, to the formal nature of the specification and design tools used, and, on the other hand, to the application of different development techniques to obtain the processors from their specifications. In this thesis we address this problem in the case of attribute grammars as the basic specification formalism.

Firstly, we present the conception of an approach aimed to facilitate the comprehension of attribute grammars fundamentals and its application to the specification of language processors. Also, we have developed a software system that supports the mentioned strategy. The strategy proposes an approach driven by exercises in which students have to emulate the semantic evaluation process of sentences according to an attribute grammar. In order to support this strategy, we have developed a software system called *Evaluators* that takes batteries of the kind of exercises described previously to automatically generate interactive simulators that students can use to solve the mentioned exercises. The system is compounded by an authoring tool, used by teachers to create the exercises, and an analysis tool, that lets teachers trace the behavior of students in the simulators generated. Regarding these simulators, the tool is able to generate two kinds of simulations: simulators based on serious games, and interactive simulators based on conventional abstract representations used in lectures. The thesis presents different assessments results of the system and the strategy with students and teachers that demonstrate the practical and educational utility of both. By last, in this thesis we abstract the process model used in the development of *Evaluators*, in order to enable the extrapolation of this approach to other specification formalisms or even other educational domains different to Compiler Construction courses.

Also, this thesis copes with the aspects related with the creation of the specifications. To this end, we have introduced a development approach for language processors driven by the specifications based on attribute grammars, which advocates the preservation of the specification during development as well as the development based on evolutionary prototypes. For that, the approach proposes a direct pattern for coding arbitrary non-circular attribute grammars as translation schemata, through conventional parser generation tools (e.g., JavaCC o CUP). The resulting codifications preserve, on one side, a direct relation with the original attribute grammar, and, on the other side, let students experiment directly with the runnable prototypes generated, as long as evolve these prototypes through different optimizations. This coding activity is supported by the use of a library called *EvLib*, which facilitates to couple a semantic evaluation process with conventional parsing models (both top-down and bottom-up ones). The approach has also been evaluated positively both by students and teachers in the field.

Finally, this thesis combines the two previous contributions through the creation of an educational environment for the development of language processors from their specifications based on attribute grammars. This environment, which is called *EvDebugger*, can generate automatically translation

schemata that implement the working prototypes of language processors from their attribute grammar-based specifications. This generation process makes use of the coding patterns of attribute grammars as translation schemata described previously, and supported by the library *EvLib*. The environment also lets students explore the semantic evaluation process adopting the educational strategy underlying *Evaluators*. Preliminary evaluations of the tool with students of Compiler Construction courses show, also, the practical utility of the tool.

Índice general

Índice de figuras

Capítulo 1: Introducción

Los conceptos y técnicas implicados en el diseño y en la implementación de lenguajes informáticos constituyen una parte esencial de la formación básica de cualquier ingeniero informático. De hecho, estos conceptos están incluidos en las principales recomendaciones curriculares para los estudios superiores en Informática [2]. Además, estos conceptos son esenciales para entender enfoques de desarrollo de software avanzados, tales como los enfoques generativos [36] [83], técnicas de desarrollo dirigido por modelos, métodos de desarrollo dirigidos por lenguajes específicos de dominio (DSLs[1]) [50] [88] [171], etc. Por ello, la mayoría de planes docentes de los títulos universitarios de Informática incluyen materias que abordan estos temas.

Tradicionalmente estos contenidos se cubren en cursos de introducción a la construcción de compiladores [5]. Dichos cursos perfilan modelos de desarrollo de software fuertemente apoyados en métodos formales de especificación, así como en técnicas sistemáticas de implementación a partir de las especificaciones resultantes. Efectivamente:

- En dichos modelos, se comienza especificando los diferentes aspectos de un lenguaje informático y de su procesamiento utilizando formalismos específicamente orientados a cada uno de los aspectos (expresiones regulares, gramáticas incontextuales, gramáticas de atributos, etc.).
- Seguidamente se aplican técnicas sistemáticas de implementación (muchas veces apoyadas en herramientas de propósito específico) para construir los procesadores de lenguaje finales.

De esta forma, el resultado final es altamente dependiente de la calidad de las especificaciones, así como de la destreza del desarrollador para transformar dichas especificaciones en implementaciones operativas. Sin embargo, muchas veces los estudiantes que se inician en estas materias desdeñan la importancia de los formalismos de especificación, al encontrar los mismos excesivamente abstractos y

[1] Domain-Specific Languages

desconectados de enfoques más convencionales al desarrollo de software, por lo que no alcanzan la comprensión y destreza necesarias para su aplicación práctica. El resultado es la confección de especificaciones incorrectas o incompletas, una transición prematura a las fases más gratificantes de implementación, una construcción *ad hoc* de los procesadores con una desconexión palpable entre la implementación final y la especificación, y, como resultado, procesadores de lenguaje de baja calidad, que, bien funcionan incorrectamente, o que, si bien funcionan correctamente, son muy difíciles de trazar, mantener y extender [158] [159].

Es de estos problemas de donde surge la motivación para realizar esta tesis, que pretende proporcionar herramientas y métodos educativos a los docentes y estudiantes de materias afines a los citados cursos introductorios de construcción de compiladores a fin de facilitar la adquisición de los conocimientos y destrezas básicos requeridos durante las fases de especificación y de implementación sistemática dirigida por especificaciones de procesadores de lenguaje.

Esta tesis ha sido desarrollada en el seno del grupo de investigación ILSA (Ingeniería de Lenguajes Software y Aplicaciones) de la Universidad Complutense de Madrid. A continuación se exponen los objetivos y planteamiento de la línea de investigación de la misma.

1.1 Objetivos y planteamiento de la línea de investigación

Tal como se ha descrito anteriormente, el correcto desarrollo de un procesador de lenguaje depende de una correcta orquestación de métodos previos de especificación formal, y de la subsecuente aplicación de técnicas sistemáticas de implementación dirigidas por dichas especificaciones. No obstante, los estudiantes habituales de la materia encuentran, en mayor o menor medida, serias dificultades para asimilar de manera satisfactoria dicha orquestación [158] [159]. Por ello, el principal objetivo de esta tesis es proporcionar métodos, estrategias y herramientas educativas para propiciar la asimilación correcta de dicha orquestación, esencial para llevar a cabo el desarrollo sistemático de los procesadores de lenguaje. Estos métodos y herramientas aprovecharán las ventajas que ofrecen las nuevas tendencias en educación y eLearning para presentar los conceptos de una forma más cercana, accesible y atractiva a los alumnos.

De esta forma, la investigación planteada en este trabajo de tesis se basa en el desarrollo de estrategias y herramientas educativas para propiciar la comprensión de los métodos de especificación y su correcto uso en el desarrollo de procesadores de lenguaje, así como en validar empíricamente su idoneidad a través de distintos experimentos llevados a cabo en cursos relacionados con dicho desarrollo. Para ello se plantean los siguientes objetivos:

- El primer objetivo se centra en desarrollar un método de enseñanza para mejorar la comprensión de los conceptos básicos involucrados en la especificación de un procesador de lenguaje, en implementar dicho método

mediante un conjunto apropiado de herramientas informáticas, y en evaluar la idoneidad pedagógica del método y de las herramientas.

- El segundo objetivo se centra en proporcionar una estrategia de desarrollo que facilite a los alumnos la implementación temprana de la especificación de un procesador de lenguaje, así como en evaluar el impacto pedagógico de la estrategia.
- Por último, el tercer objetivo se centra en proporcionar un entorno de desarrollo de procesadores de lenguaje basado en la especificación que integre tanto los aspectos de comprensión de las especificaciones como los aspectos de desarrollo de los procesadores a partir de dichas especificaciones, así como en evaluar dicho entorno desde el punto de vista de su uso educativo.

Para abordar el primer objetivo se ha elegido el formalismo de las *gramáticas de atributos* [89] [90] [120] como formalismo básico utilizado para especificar los procesadores. Esta elección se justifica, por una parte, por el amplio uso de este formalismo en muchos cursos de introducción a los procesadores de lenguaje [5] [100]. Por otra parte, se justifica porque este es el formalismo utilizado en los cursos en los que se han validado empíricamente las propuestas realizadas. De esta forma, los trabajos relativos a la consecución de este objetivo se han enfocado en los aspectos de *comprensión* del formalismo elegido, las gramáticas de atributos, con el fin de ayudar a los estudiantes a comprender los conceptos y mecanismos básicos de evaluación semántica que subyacen al mismo. Con ello, el propósito ha sido mejorar la asimilación de dichos conceptos y mecanismos en las primeras fases del proceso de enseñanza-aprendizaje. La hipótesis aquí ha sido que dicha asimilación temprana contribuirá a ayudar a los alumnos a utilizar mejor el formalismo para especificar las tareas básicas de procesamiento dirigido por sintaxis en las que se apoya la construcción de los procesadores, y, como consecuencia, a mejorar el proceso global de desarrollo de los procesadores. Para ello se ha diseñado una estrategia para la enseñanza-aprendizaje del proceso de evaluación semántica asociado con las gramáticas de atributos, y se han desarrollado distintos sistemas informáticos basados en dicha estrategia. El denominador común de dichos sistemas es promover un *enfoque generativo* y *dirigido por problemas*, que involucra, de manera coordinada, a docentes que plantean ejercicios de evaluación semántica a resolver, así como a estudiantes que resuelven dichos ejercicios. Así mismo, y aunque el trabajo realizado se ha enfocado completamente hacia las gramáticas de atributos, se ha realizado también un esfuerzo importante en abstraer el proceso de construcción de dichos sistemas, de forma que el método seguido pueda extrapolarse fácilmente para tratar otro tipo de formalismos de especificación.

El segundo objetivo se ha abordado mediante la creación de una estrategia de desarrollo de procesadores de lenguaje apoyada por herramientas convencionales de generación de traductores (CUP, JavaCC, etc.) que permita a los estudiantes implementar de manera rápida sus especificaciones basadas en gramáticas de atributos mediante modelos de traducción típicos. De esta forma, el trabajo relativo a este objetivo se ha centrado más en la mejora de los aspectos de *creación* efectiva de los procesadores por parte de los estudiantes, promoviendo, para ello, un proceso sistemático que propugna la codificación directa de la especificación en la herramienta. El resultado es un primer prototipo ejecutable del procesador de

lenguaje implementado en una herramienta de construcción de procesadores de lenguaje convencional, que preserva fielmente la especificación. Dicho prototipo puede servir como base para realizar distintas transformaciones y mejoras sistemáticas orientadas a añadir eficiencia al procesador resultante.

El tercer objetivo sirve, finalmente, como nexo de unión a los dos primeros. Para ello se ha implementado un entorno de desarrollo para la generación de procesadores de lenguaje que es capaz de obtener su implementación sobre herramientas convencionales de generación de dichos procesadores a partir de su especificación en forma de gramática de atributos, automatizando, de esta forma, los patrones de codificación manual de las especificaciones ideados durante la consecución del objetivo dos. Además, para facilitar la comprensión de sus propios diseños y ofrecer soporte a la depuración de los mismos, el entorno ofrece un sofisticado depurador visual, basado en la estrategia de comprensión desarrollada durante la ejecución del objetivo uno.

Así mismo, y tal y como se establece en los distintos objetivos, se ha puesto especial énfasis en la evaluación empírica de cada una de las estrategias y herramientas creadas. Para ello se han llevado a cabo diferentes estudios y experiencias entre docentes y estudiantes de asignaturas relativas a la construcción de Procesadores de Lenguaje, tanto en la *Universidad Complutense de Madrid* como en la *Universidade do Minho* (en Portugal). Con los resultados obtenidos de cada una de estas experiencias se ha podido comprobar la valoración de estudiantes y docentes de las distintas propuestas, la utilidad percibida de cada uno de los artefactos ideados, y su eficacia educativa.

1.2 Estructura de la memoria

Este trabajo de tesis, que se presenta como una recopilación de publicaciones, desarrolla, aparte de esta introducción, los siguientes seis capítulos:

- El capítulo 2 presenta una revisión de conceptos implicados en este trabajo de tesis. Para ello comienza describiendo las estrategias educativas más ampliamente utilizadas para enseñar las materias relativas a la construcción de procesadores de lenguaje. Seguidamente, revisa diferentes herramientas software educativas para la enseñanza de conceptos relativos a la Informática, y más en concreto, relacionadas con la enseñanza de conceptos relativos a la construcción de procesadores de lenguaje. De igual modo, se estudian diferentes enfoques basados en juegos educativos orientados a la enseñanza en los citados campos: Informática y Procesadores de Lenguaje. Finalmente, se revisan diferentes técnicas y herramientas para el desarrollo y construcción de procesadores de lenguaje.
- El capítulo 3 detalla los objetivos principales de esta tesis y la planificación del trabajo ideada para acometer dichos objetivos.
- El capítulo 4 detalla las diferentes contribuciones aportadas por los artículos que acompañan a este trabajo de tesis a la hora de acometer los objetivos planteados.

- El capítulo 5 describe las conclusiones finales de este trabajo de tesis, y plantea las diferentes líneas de trabajo futuro que surgen de los resultados obtenidos al acometer los objetivos planteados.
- El capítulo 6 contiene los diferentes artículos que componen este trabajo de tesis, en su versión original.
- Finalmente, el capítulo 7 presenta un resumen extendido en inglés de los capítulos que componen este trabajo de tesis, para hacerlo más accesible a potenciales lectores no hispano hablantes.

1.3 Resumen de las contribuciones

Los objetivos planteados en este trabajo de tesis cristalizan en diferentes contribuciones que se desarrollarán a lo largo de los siguientes capítulos. En esta sección se resumen estas contribuciones y los artículos en los que se han publicado los resultados asociados a las mismas (véase el capítulo 4 para una descripción más detallada).

La primera contribución es una estrategia para la adquisición de conceptos básicos del formalismo de las gramáticas de atributos en etapas tempranas del aprendizaje de los alumnos soportada por un sistema software educativo capaz de generar diferentes tipos de simuladores interactivos (basados en juegos serios, y basados en visualizaciones). Además, la estrategia y el software de apoyo han sido evaluados con docentes y estudiantes de un curso de Procesadores de Lenguaje para demostrar su utilidad educativa. Finalmente, con el fin de aumentar la aplicabilidad de la estrategia a otros formalismos de especificación, se ha creado un modelo de proceso para el desarrollo de sistemas software educativos en el marco de estrategias educativas similares. Los artículos que registran los resultados de esta contribución son los siguientes [135] [137] [138] [139] [141] [142]:

- Rodríguez-Cerezo D., Sarasa-Cabezuelo A., Gómez-Albarrán M., Sierra-Rodríguez J.L. User-Centered Development of Generative Educational Systems for Computer Engineering: The Evaluators Case Study. International Journal of Engineering Education, 31(3): 751–763, 2015. (Índice de impacto JCR en 2014: 0.582).
- Rodríguez-Cerezo, D., Gómez-Albarrán, M., Sierra, J. L. A Process Model for the Generative Production of Interactive Simulations in Engineering Education. En TEEM'13: Proceedings of the First International Conference on Technological Ecosystem for Enhancing Multiculturality, 95-103. 2013.
- Rodriguez-Cerezo, D., Gomez-Albarrán, M., Sierra, J. L. From Collections of Exercises to Educational Games: A Process Model and a Case Study. En ICALT'11: Proceedings of the 11th IEEE International Conference on Advanced Learning Technologies, 282-284. 2011. (Conferencia clasificada como CORE B).
- Rodriguez-Cerezo, D., Gómez-Albarrán, M., Sierra-Rodríguez, J. L. Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts. En ITiCSE'13: Proceedings of the

18th ACM Conference on Innovation and Technology in Computer Science Education, 28-33. 2013. (Conferencia clasificada como CORE A).

- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J. L. Serious Games in Tertiary Education: A Case Study Concerning the Comprehension of Basic Concepts in Computer Language Implementation Courses. Computers in Human Behavior, 31: 558-570. 2014. (Índice de impacto JCR en 2014: 2.694).
- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., & Sierra, J.L. Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game-based Approach. En SIIE'12: Proceedings of the International Symposium on Computers in Education, 1-6. 2012.

La segunda contribución consiste en una estrategia de desarrollo de procesadores de lenguaje que, a partir de gramáticas de atributos no circulares arbitrarias, permite obtener sistemáticamente implementaciones basadas en esquemas que traducción que preservan una correspondencia directa entre la implementación final y la especificación de partida. Esta estrategia está acompañada por un software que da soporte a la estrategia permitiendo la especificación de las ecuaciones semánticas en el esquema de traducción y la emulación del proceso de cálculo de las gramáticas de atributo en el traductor resultante. Además, se han registrado resultados empíricos de experiencias de valoración realizadas con docentes y estudiantes para validar la estrategia. Los artículos asociados a esta contribución son los siguientes [136] [143] [144]:

- Rodríguez-Cerezo D., Sierra, J.L. Introducing a Design-Preserving Implementation Strategy in a Compiler Construction Course. En SIIE'13: Actas del XV Simposio Internacional de Informática Educativa, 24-29. 2013.
- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools. Computer Science and Information Systems, 9(3): 983-1017. 2012. (Índice de impacto JCR en 2012: 0.549).
- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. Implementing Attribute Grammars Using Conventional Compiler Construction Tools. En FedCSIS'11: Proceedings of the Federated Conference on Computer Science, 855-862. 2011.

Por último, la tercera contribución es una plataforma educativa de construcción de procesadores de lenguaje a partir de especificaciones descritas mediante el formalismo de las gramáticas de atributos. Esta plataforma proporciona un depurador visual enriquecido basado en la estrategia educativa resultante de la primera contribución, que emula el proceso de cálculo intrínseco a las gramáticas de atributos. Por otro lado, la plataforma es capaz de codificar las especificaciones de entrada, en forma de gramática de atributos, como esquemas de traducción haciendo uso de los patrones de desarrollo resultantes de la segunda contribución de este trabajo de tesis. El artículo asociado a esta tercera contribución es el siguiente [140]:

Capítulo 1:
Introducción

Rodríguez-Cerezo, D., Henriques, P. R., Sierra, J. L. Attribute Grammars Made Easier: EvDebugger a Visual Debugger for Attribute Grammars. En SIIE'14: Proceedings of International Symposium on Computers in Education, 23-28. 2014.

Capítulo 2: Estado de la cuestión

Como ya se ha indicado en el capítulo anterior, esta tesis aborda la mejora del proceso de enseñanza-aprendizaje en materias relativas al diseño y a la implementación de lenguajes informáticos en relación con los aspectos de especificación (en particular, aquellos relativos al uso de formalismos declarativos para la descripción de tareas de procesamiento de lenguaje dirigidas por la sintaxis) y de implementación sistemática de procesadores a partir de especificaciones. De esta forma, en este capítulo se revisan los elementos más relevantes de cara a llevar a cabo la contextualización del trabajo de tesis presentado.

El capítulo comienza revisando los aspectos más relevantes que, para esta tesis, entraña la enseñanza de las materias relacionadas con el desarrollo de procesadores de lenguaje (apartado 2.1). A continuación, y dada la relevancia que tanto las simulaciones como los juegos educativos cobran en esta tesis en relación con la comprensión de los conceptos básicos involucrados en la especificación de un procesador de lenguaje, se revisan las principales características de dichas simulaciones (apartado 2.2) y videojuegos educativos (apartado 2.3), haciendo especial énfasis en la enseñanza de la Informática en general, y de la materia de Procesadores de Lenguaje en particular. El apartado 2.4 revisa los aspectos más relevantes para esta tesis de las herramientas de desarrollo de procesadores de lenguaje, así como de los usos educativos de las mismas. Para finalizar, el apartado 2.5 concluye este análisis del estado de la cuestión, exponiendo las conclusiones más relevantes de cara al desarrollo de la tesis.

2.1 Enseñanza de la materia de Procesadores de Lenguaje

El objeto de esta tesis es contribuir a mejorar la enseñanza-aprendizaje de los aspectos relativos a la especificación e implementación sistemática de procesadores de lenguaje. Con ello se trata de aliviar las problemática que, históricamente, rodea a la enseñanza-aprendizaje de las materias relacionadas con el diseño de lenguajes informáticos y la implementación de sus procesadores:

- Los cursos sobre construcción de procesadores de lenguaje suelen contemplar el uso coordinado de una gran cantidad de conceptos teóricos y técnicas sistemáticas de desarrollo. Es muy habitual que los docentes no sean capaces de cubrir en profundidad cada uno de estos conceptos y técnicas, proporcionando una visión superficial de los mismos, lo que dificulta su aplicación al diseño y desarrollo real de un procesador [4].
- Los alumnos, además, deben completar con éxito el desarrollo de un procesador para un lenguaje informático (normalmente, un lenguaje de programación reducido). Es en este punto donde ellos encuentran la mayoría de problemas, siendo incluso la complejidad de esta tarea motivo de abandono de la asignatura por parte de muchos alumnos [66].
- Los estudiantes no son capaces de conectar adecuadamente los conceptos teóricos y formalismos explicados en clase, con las técnicas y metodologías de desarrollo. Esto provoca que los alumnos desarrollen procesadores que no funcionan adecuadamente, debido muchas veces a unas especificaciones incorrectas [159].
- Los estudiantes consideran la mayoría de conceptos que componen la asignatura como demasiado complejos, debido en gran medida a su naturaleza formal. Esto provoca a los estudiantes una pérdida de motivación a la hora de abordar la asignatura, llegando incluso a que estos perciban los conceptos impartidos en el curso como poco útiles en su vida laboral [174], a pesar de que los conceptos presentados en estas materias son fundamentales para entender, y aplicar, los nuevos enfoques de desarrollo de software dirigido por modelos y por lenguajes específicos de dominio [50] [88].

A fin de paliar la problemática asociada a la misma, tradicionalmente los docentes de la materia de Procesadores de Lenguaje han aplicado diferentes estrategias pedagógicas destinadas a facilitar a sus estudiantes la asimilación de la gran cantidad de conceptos, técnicas y métodos que componen la disciplina. Estas estrategias hacen especial énfasis en la parte práctica del proceso de enseñanza/aprendizaje como un componente esencial para una asimilación satisfactoria de la materia. Tres de las estrategias más representativas, que se detallan en las siguientes secciones, son:

- Estrategia basada en el desarrollo de un procesador para un lenguaje informático completo (sección 2.1.1).
- Estrategia centrada en la construcción de procesadores para pequeños lenguajes orientados a ilustrar distintos patrones de procesamiento (sección 2.1.2).

- Estrategia basada en el análisis y la depuración de un procesador real (sección 2.1.3).

El apartado finaliza, así mismo, con una discusión de estas estrategias (sección 2.1.4).

2.1.1 Estrategia basada en el desarrollo de un procesador para un lenguaje completo

Esta estrategia es la más empleada por una gran parte de docentes. Esta estrategia consiste en proponer a los alumnos el desarrollo de un procesador para un lenguaje informático completo, que traducirá los códigos fuente a un lenguaje objeto adecuado. A este respecto:

- Muchos enfoques se basan en elegir lenguajes de programación como lenguajes fuente. Existe un amplio abanico de lenguajes de programación reducidos prototípicos para esta estrategia (COOL [6] o MINIML [19] son dos ejemplos en dicho abanico). Estos lenguajes comparten, normalmente, las siguientes características: incluir un par de tipos básicos (como números naturales y booleanos), un conjunto de operaciones básicas para esos tipos, estructuras de control (por ejemplo bucles o condicionales), tipos definidos por el programador (vectores, registros, etc.), y un mecanismo de abstracción (como procedimientos y/o funciones).
- Otros enfoques se centran en lenguajes menos convencionales, que los docentes suelen proponer a sus alumnos con el fin de propiciar un atractivo extra, debido a la finalidad u orientación de los lenguajes, lo que permite motivar a dichos alumnos. Entre dichos lenguajes se encuentran lenguajes de representación de grafos como [177] (un lenguaje que proporciona la suficiente expresividad para definir grafos que podrán ser dibujados por el procesador de lenguaje pedido a los alumnos), lenguajes de dibujo de figuras como [145] (un lenguaje para especificar gráficos construidos a partir de figuras simples, de forma tal que el correspondiente procesador se encargue de realizar el dibujo especificado), o lenguajes de programación de robots como [181] (un lenguaje para especificar acciones sobre un robot). La principal ventaja que presenta este tipo de lenguajes, en comparación a otros lenguajes informáticos más convencionales, es su atractivo para los alumnos, lo que puede motivarles y hacerles sentirse más implicados durante el desarrollo del proyecto, y por lo tanto mejorar su aprendizaje.

Independientemente del lenguaje escogido por el docente, a la hora de implementar el procesador es posible observar también diferentes opciones:

- Algunos profesores optan por exigir a los alumnos que realicen la implementación partiendo desde cero, tal y como se muestra en [48][153].
- En algunas ocasiones, los docentes proporcionan a sus alumnos un conjunto básico de utilidades [18] como componentes para manejar la tabla de símbolos, generación de código, etc.
- Otros docentes abogan por centrarse en enseñar a los alumnos las herramientas existentes para programar procesadores de lenguaje, permitiendo a sus alumnos desarrollar sus procesadores usando

herramientas de generación automática, como YACC [154], Bison [97] o JavaCC [91](en [39][102] pueden encontrarse ejemplos de aplicación de esta estrategia). Esta estrategia de implementación permite a los alumnos centrarse en los aspectos básicos del procesamiento de lenguaje, y no en los detalles de más bajo nivel.

2.1.2 Estrategia basada en pequeños proyectos de procesamiento de lenguaje

El principal problema de la estrategia basada en el desarrollo de un procesador para un lenguaje completo es que, aun siendo el lenguaje soportado simple, la implementación de dicho procesador puede no ser trivial. Efectivamente:

- Durante las primeras fases del desarrollo los alumnos pueden encontrar problemas con la especificación de los distintos aspectos (léxico, sintaxis, restricciones semánticas, etc.), lo que puede suponer una gran demora en el proceso de implementación, pudiendo conllevar incluso que los alumnos no completen con éxito el proyecto.
- Existe una desincronización entre el avance de las clases teóricas y la realización de la práctica, ya que determinados conceptos o técnicas necesarios para la realización de la práctica no son impartidos hasta bien avanzado el curso. Esto limita aún más el tiempo que los alumnos disponen para terminar el proyecto.

Teniendo en cuenta estos problemas, se ha propuesto la estrategia basada en pequeños proyectos de procesamiento de lenguaje. En esta estrategia, según se va avanzando en las lecciones durante el curso, los profesores proponen pequeños proyectos centrados en lenguajes muy simples, directamente relacionados con los conceptos vistos en clase. Además, los docentes suelen proporcionar a los alumnos partes del procesador pedido ya implementadas (por ejemplo, el analizador léxico o el sintáctico) para concentrar la atención de los mismos en los conceptos relacionados con el proyecto, que se habrán introducido, así mismo, recientemente en clase. En [95] [157] se analizan los beneficios de esta estrategia.

2.1.3 Estrategia basada en el análisis y depuración de un procesador de lenguaje real

Esta estrategia surge en respuesta al principal problema que muestran las estrategias basadas en la realización de proyectos de procesamiento para lenguajes de programación reducidos, o las basadas en pequeños proyectos de procesamiento de lenguajes: estos proyectos no son suficientes para enseñar todas las técnicas y conceptos implicados en el desarrollo de procesadores para muchos lenguajes informáticos reales. Por ello, en [178] se propone ilustrar los conceptos y técnicas que componen la materia valiéndose de las herramientas actuales de depuración sobre un procesador profesional. En concreto, en el trabajo citado, se utiliza un compilador de C# para realizar la depuración. Las clases se organizan en sesiones de laboratorio donde los docentes usan *breakpoints* para, a continuación, visualizar las variables más significativas involucradas en el procesamiento (por ejemplo, la tabla de símbolos, o el código generado). Los docentes irán depurando

el compilador para mostrar los pasos que va realizando este y mostrar a los alumnos cómo se aplican las técnicas y métodos enseñados en las clases teóricas. Durante estas sesiones, los docentes no depuran completamente todos los procesos internos, sino que se centran en las partes más importantes e inmediatamente relacionadas con los conceptos impartidos en clase.

2.1.4 Discusión

Las diferentes metodologías para la enseñanza de Procesadores de Lenguaje expuestas adoptan diferentes enfoques pedagógicos, mostrando estas diferentes ventajas y desventajas entre sí. En concreto:

- La implementación de un procesador para un lenguaje completo obliga a los alumnos a desarrollar un procesador para un lenguaje informático utilizando directamente las técnicas y formalismos mostrados en las clases teóricas. Por otro lado, esta clase de proyectos ofrece a los alumnos una buena oportunidad para aplicar sus conocimientos en gestión de proyectos e ingeniera del software por la magnitud del proyecto y la calidad exigida. Sin embargo, la principal desventaja de esta metodología es que el orden de complejidad del desarrollo puede sobrepasar las destrezas del alumno novel promedio en el diseño e implementación de lenguajes informáticos.
- La metodología basada en la realización de pequeños proyectos de procesamiento de lenguaje propone a los alumnos aplicar los conceptos vistos en clase directamente en el desarrollo de procesadores de micro-lenguajes. Ya que los proyectos se proponen inmediatamente después de impartir las clases teóricas relacionadas, los alumnos tienen mayor facilidad para retener los conceptos impartidos. Sin embargo, al realizar estos proyectos pequeños, los alumnos pierden la perspectiva más global que proporciona la realización de un proyecto de gran envergadura.
- Finalmente, en el análisis y depuración de un procesador de lenguaje real los docentes muestran cómo se aplican los conceptos y métodos explicados en el aula teóricamente, mediante la depuración de un procesador de lenguaje real. Sin embargo, estos procesadores profesionales han sido optimizados y resulta más difícil identificar las técnicas y algoritmos expuestos en clase dentro del código depurado. Además, los alumnos se vuelven meros espectadores, no pudiendo desarrollar adecuadamente las habilidades de implementación y especificación requeridas.

Independientemente de sus ventajas e inconvenientes, estas tres estrategias tienen un denominador común: se centran fundamentalmente en aspectos relativos a la implementación. Teniendo en cuenta que los aspectos de especificación son igual de importantes, o incluso más, y necesarios para el correcto desarrollo del procesador de lenguaje, esta fuerte orientación hacia los aspectos de implementación, en detrimento de los aspectos de especificación, provoca que los alumnos encuentren serias dificultades para entender los fundamentos de las especificaciones formales mostradas en clase, y que son necesarias para el correcto desarrollo de los procesadores propuestos en las metodologías descritas.

2.2 Simulaciones educativas

En base a la importancia que adquieren los aspectos relativos a la especificación en las materias relacionadas con los procesadores de lenguaje, en esta tesis se considera que la mejora en la comprensión de dichos aspectos es clave de cara a mejorar el proceso global de enseñanza-aprendizaje de dichas materias. Para ello, esta tesis promueve, entre otros instrumentos, el uso de distintos tipos de simulaciones interactivas para facilitar la comprensión de los conceptos básicos relativos a las especificaciones formales en el ámbito del diseño y la implementación de lenguajes informáticos (gramáticas de atributos, en particular). Es por ello que, en este apartado, se revisan los aspectos más relevantes de este tipo de simulaciones.

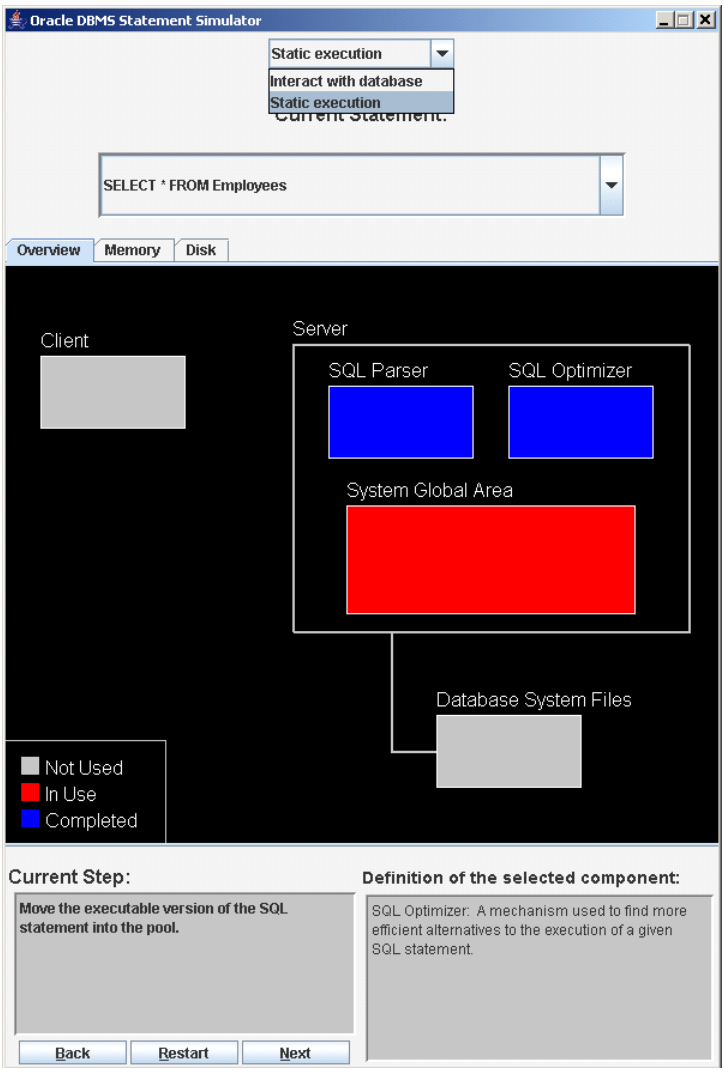


Figura 1. Captura del simulador del sistema de consultas de bases de datos (obtenida de [10])

Las simulaciones educativas facilitan la comprensión de conceptos por parte de los estudiantes, así como ayudan a motivar a los alumnos, estimulan su curiosidad intelectual, sentido del control personal, y su perseverancia [21] [79]. Ampliamente utilizadas en la enseñanza de diferentes campos, como la aviación [161] o la medicina [103], las simulaciones presentan a los estudiantes entornos virtuales interactivos donde pueden aprender mediante la experimentación e

interacción con el entorno presentado. Normalmente, los diseñadores de estos simuladores crean entornos virtuales acordes a las necesidades de los estudiantes, lo que orienta y focaliza el hilo educativo que los alumnos seguirán al usar el simulador. Aun así, el entorno virtual de simulación no tiene por qué estar basado necesariamente en entornos reales: es el caso de aquellos simuladores centrados en modelos teóricos. Es por ello que los simuladores resultan de utilidad para enseñar el funcionamiento de diferentes algoritmos y otros conceptos relacionados con la Informática.

En las siguientes secciones se describen diferentes simuladores y herramientas basadas en visualización para mejorar el aprendizaje de distintas materias implicadas en la enseñanza de la Informática (sección 2.2.1), y en concreto, aquellas orientadas a las materias relativas al diseño e implementación de lenguajes informáticos (sección 2.2.2). Se discuten, así mismo, los aspectos más relevantes de estos enfoques de cara al desarrollo de esta tesis (sección 2.2.3).

2.2.1 Simulación en la Informática

Siendo conscientes del poder de la simulación como herramienta para mejorar el aprendizaje de los alumnos, muchos docentes hacen uso de simulaciones educativas. En especial, los docentes de Informática son más conscientes de estas capacidades a la hora de presentar el funcionamiento de diferentes algoritmos y formalismos y, además, son capaces de desarrollar sus propias aplicaciones que ofrecen al resto de docentes. Algunos ejemplos representativos de este tipo de herramientas en el campo de la educación superior en Informática son:

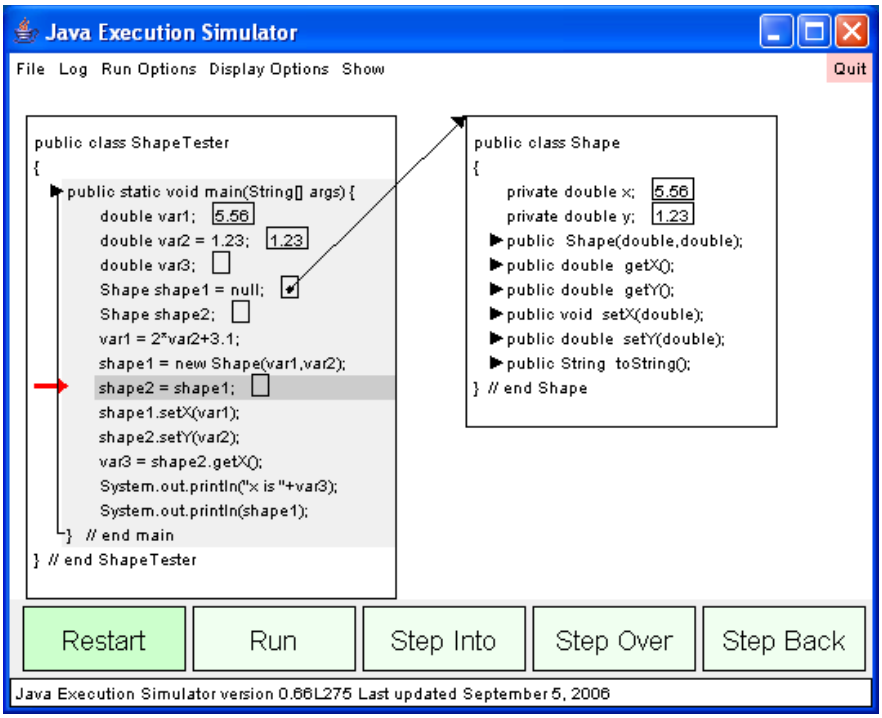


Figura 2. Captura de JES (obtenida de [131])

- *Query Simulator System* [10]. Esta es una herramienta basada en visualizaciones desarrollada para el ámbito de las bases de datos. El principal objetivo de esta herramienta es ilustrar el funcionamiento interno

de las bases de datos al llevar a cabo diferentes tipos de consultas. El software representa los distintos componentes, lógicos y físicos, de las bases de datos SQL y, mediante cambios de estado de estos componentes, se representan los procesos internos que ocurren al realizar una consulta SQL. La herramienta se comporta como un simulador proporcionando características para ejecutar paso a paso una serie de tipos de consultas SQL predefinidos (INSERT, UPDATE, etc.). La figura 1 muestra una captura de este simulador del comportamiento de las bases de datos.

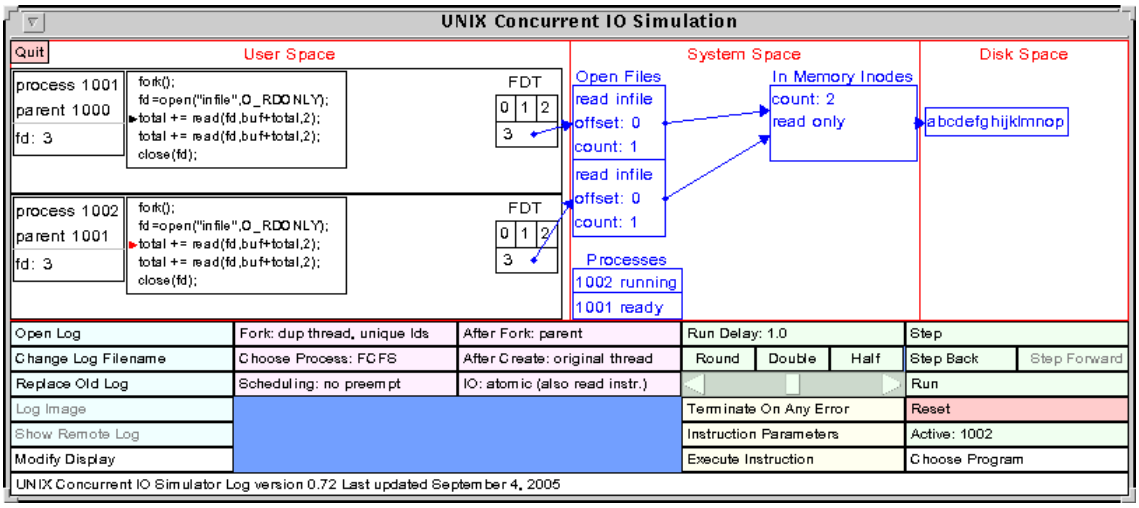


Figura 3. Captura del simulador de entrada/salida concurrente (tomada de [132])

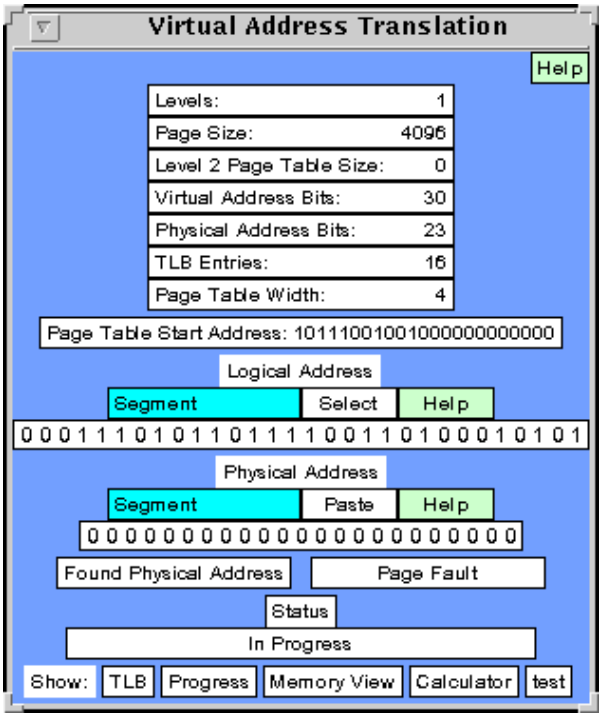


Figura 4. Captura del simulador de traducción de direcciones (tomada de [133])

- *Java Execution Simulator (JES)* [131]. Este es un simulador de la ejecución de programas Java dirigido a las primeras fases de un curso de programación orientada a objetos. La herramienta representa las diferentes clases involucradas en un programa Java, y cómo se relacionan dichas clases

entre sí. La figura 2 muestra una captura de la herramienta mostrando una relación entre dos clases.

- *UNIX Concurrent I/O Simulator* [132]. Este es un simulador de código C que, mediante visualizaciones, representa la lectura y escritura concurrente de archivos. En este caso, las visualizaciones se realizan utilizando tablas que representan los ficheros, i-nodos de memoria y la lista de procesos abiertos. Aparte de la utilidad educativa mostrada en [132], este simulador puede ser usado como un depurador orientado a la lectura/escritura concurrente en C. La figura 3 muestra la pantalla principal de este simulador.
- *Address Translation Simulator* [133]. Esta es una herramienta de simulación que muestra cómo funciona internamente la traducción de direcciones en los sistemas operativos. En [133] su autor defiende que los ejemplos usados en clase no son suficiente, por su simplicidad, y este simulador permite a los alumnos explorar exhaustivamente estos procesos. Además, es capaz de registrar la actividad realizada por los alumnos y enviarla al docente vía e-mail, para que este pueda analizarla y ofrecer una realimentación adecuada a los estudiantes. La figura 4 presenta una captura de la herramienta. Cabe destacar, así mismo, que gracias a la capacidad de registrar la actividad de los alumnos, los docentes pueden elaborar orientaciones dirigidas a los problemas y necesidades de los alumnos.

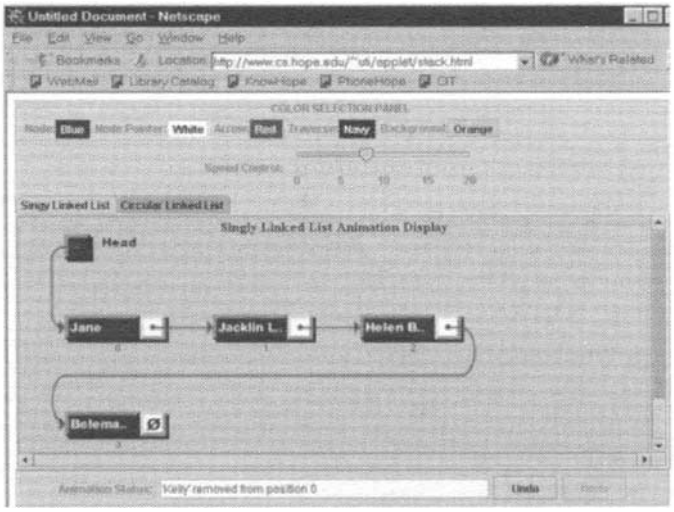


Figura 5. Captura de la visualización de la lista enlazada producida por la biblioteca de anotación de listas enlazadas (extraída de [40])

- *Animation of linkedlists in Java* [40]. Esta es una biblioteca Java que los alumnos usarán, en lugar de las listas enlazadas predefinidas en Java, para dotar a los programas creados de un visualizador de esta estructura. Su autor defiende en [40] que la biblioteca es ideal para mostrar el funcionamiento de esta estructura en las clases teóricas, como una ayuda adicional a la depuración de los programas que utilicen esta estructura de datos. La figura 5 muestra una de las visualizaciones creadas con esta biblioteca Java.

La principal característica de todos los simuladores analizados, así como otros simuladores de naturaleza similar [30][169], es que presentan de una forma atractiva los conceptos a los que están orientados, permitiendo a los alumnos descubrir y aprender dichos conceptos a través de la experimentación y la

exploración. Normalmente, la interacción ofrecida varía en función de los conceptos a enseñar y el grado de implicación que se busque en el alumno: desde simuladores como JES o *Query Simulator System* que permiten a los alumnos observar el funcionamiento de los programas Java o consultas SQL que ellos mismos desarrollan, hasta el *Address Translator Simulator* que emula paso a paso el comportamiento de cada uno de los componentes que intervienen en el proceso de traducción de direcciones. También se puede observar que estos simuladores muestran la información valiéndose de las visualizaciones y las animaciones. Así mismo, se identifican dos tipos recurrentes de visualizaciones en los ejemplos descritos: representaciones basadas en tablas donde se describe la información relevante (como en *Address Translator Simulator*), o representaciones abstractas de los componentes implicados en la simulación (como en *Query Simulator System*).

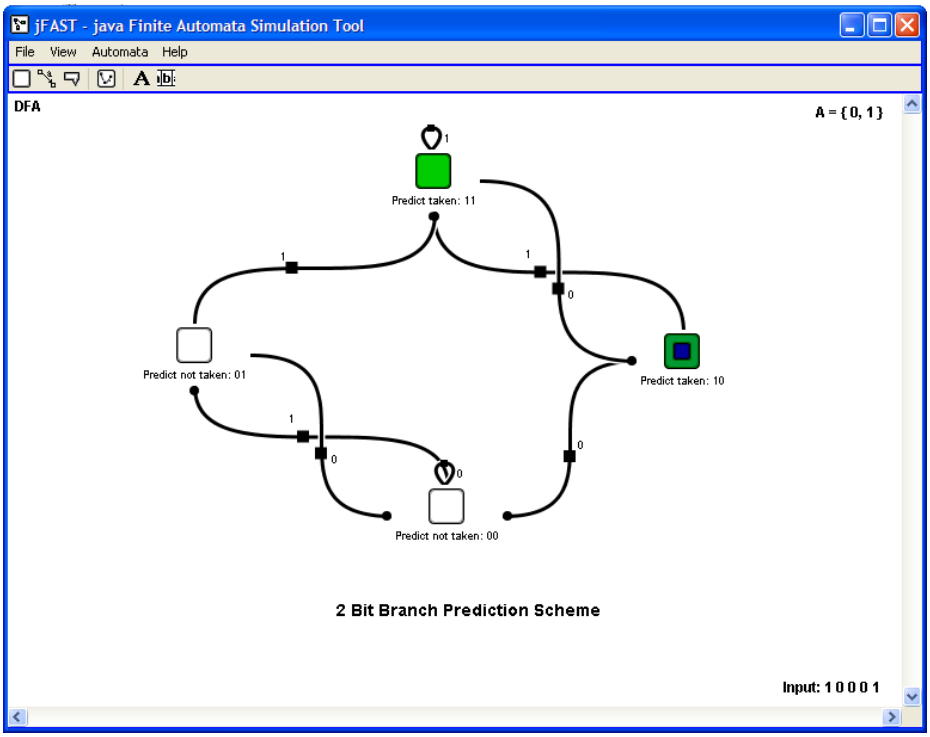


Figura 6. Captura de jFAST (tomada de [179])

2.2.2 Simulación para la enseñanza de Procesadores de Lenguaje

La materia de Procesadores de Lenguaje es susceptible también de beneficiarse de las simulaciones educativas. De esta forma, en esta sección se describen diferentes herramientas software basadas en simulaciones orientadas a mejorar el proceso de enseñanza-aprendizaje de esta materia. Para ello, estas herramientas se categorizan en tres grupos diferentes: *simuladores de máquinas teóricas*, *simuladores de algoritmos de análisis* y *simuladores de otros aspectos del procesamiento*.

2.2.2.1 Simuladores de máquinas teóricas

Estos simuladores nacen de la necesidad de presentar las maquinas teóricas implicadas en el procesamiento de lenguaje, y sus algoritmos, de una forma dinámica, atractiva e interactiva. Normalmente, estos conceptos son presentados a los alumnos haciendo uso de la pizarra o de diapositivas, y muchas veces los alumnos no son capaces de entender el funcionamiento interno de los artefactos presentados, ya que normalmente encuentran las explicaciones tediosas y difíciles de seguir. No obstante, gracias a estos simuladores, los alumnos pueden interactuar con estas máquinas y aprender de una forma más experimental y atractiva. Como ejemplos de este tipo de simuladores pueden citarse los dos siguientes:

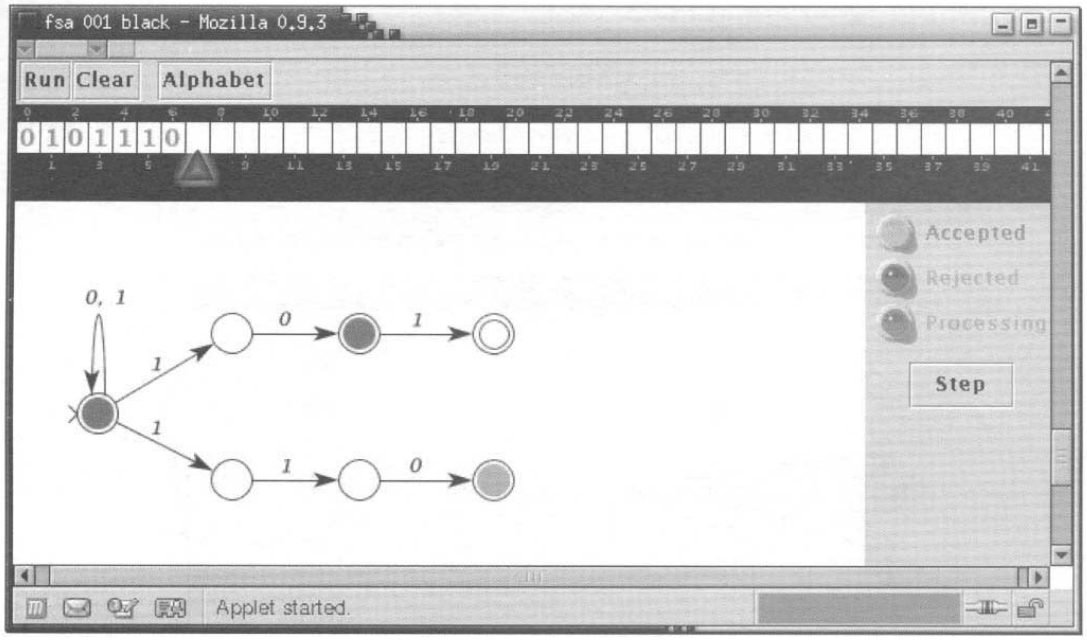


Figura 7. Captura del simulador de autómatas de estados finitos (extraída de[65])

- jFAST[179], un simulador implementado en Java y centrado en mostrar el funcionamiento de diferentes autómatas y máquinas teóricas, como autómatas finitos, autómatas con pila, máquinas de Turing, etc., mediante una interfaz sencilla y de fácil acceso para los alumnos. El simulador cuenta con un editor visual para la creación de autómatas que acerca la creación de estos artefactos a los alumnos. La figura 6 presenta una captura del editor visual de autómatas de jFAST. El simulador ha sido bien valorado por docentes y estudiantes en los trabajos descritos en [179], haciendo los estudiantes especial énfasis en la facilidad de uso del editor visual de autómatas.
- *Simulador de Autómatas de Estados Finitos* [65]. Esta es una herramienta muy similar a jFAST, que permite crear y simular diferentes tipos de autómatas y máquinas teóricas utilizando varios algoritmos propios de este campo. La principal diferencia que presenta frente a jFAST, y que la convierte en una herramienta muy útil, es que es capaz de comparar dos autómatas extrayendo las diferencias encontradas en el

análisis [65]. Esta funcionalidad es especialmente útil para los alumnos, ya que serán capaces de comparar los autómatas que ellos propongan para un determinado problema con la solución propuesta por el docente, ver sus errores y corregirlos. La figura 7 muestra una captura del simulador en funcionamiento. A pesar de esta característica, en [64] se describe un estudio realizado con alumnos que demostró que la herramienta no presenta una eficacia educativa superior al método tradicional de enseñanza de estas máquinas teóricas, aunque sí un incentivo extra para los alumnos.

Aunque la utilidad y adecuación de estas herramientas y de otras similares, como [134], para la enseñanza-aprendizaje de las materias afines a la construcción de procesadores de lenguaje es innegable, dichas herramientas están centradas en los modelos teóricos de artefactos relativos a la implementación (v.g., autómatas), en lugar de en los formalismos de especificación que preceden a dichos artefactos (v.g., expresiones regulares, gramáticas).

2.2.2.2 Simuladores de algoritmos de análisis

Este conjunto de herramientas permiten visualizar el funcionamiento de diferentes tipos de algoritmos de análisis sintáctico-semántico. De esta forma, los estudiantes son capaces de comprender el funcionamiento de estos algoritmos gracias a las representaciones de las diferentes estructuras implicadas en estos procesos y la simulación de los algoritmos. Los estudiantes pueden emular los algoritmos de análisis y observar cómo las estructuras varían conforme estos avanzan. Los siguientes son tres ejemplos representativos de este tipo de herramientas:

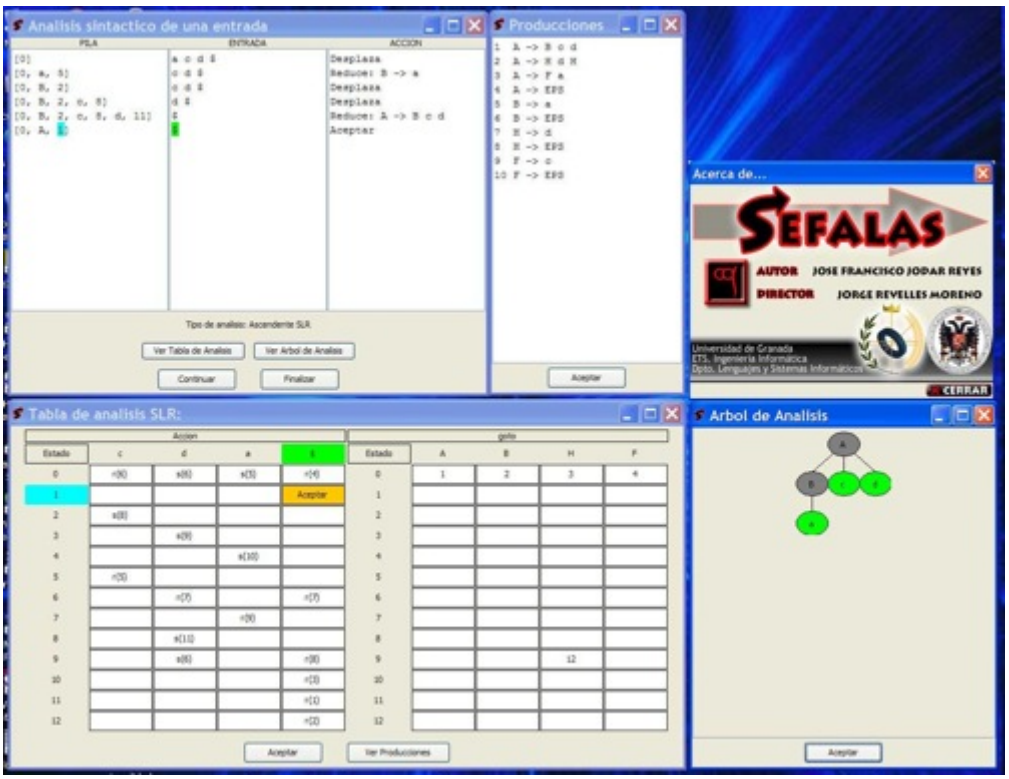
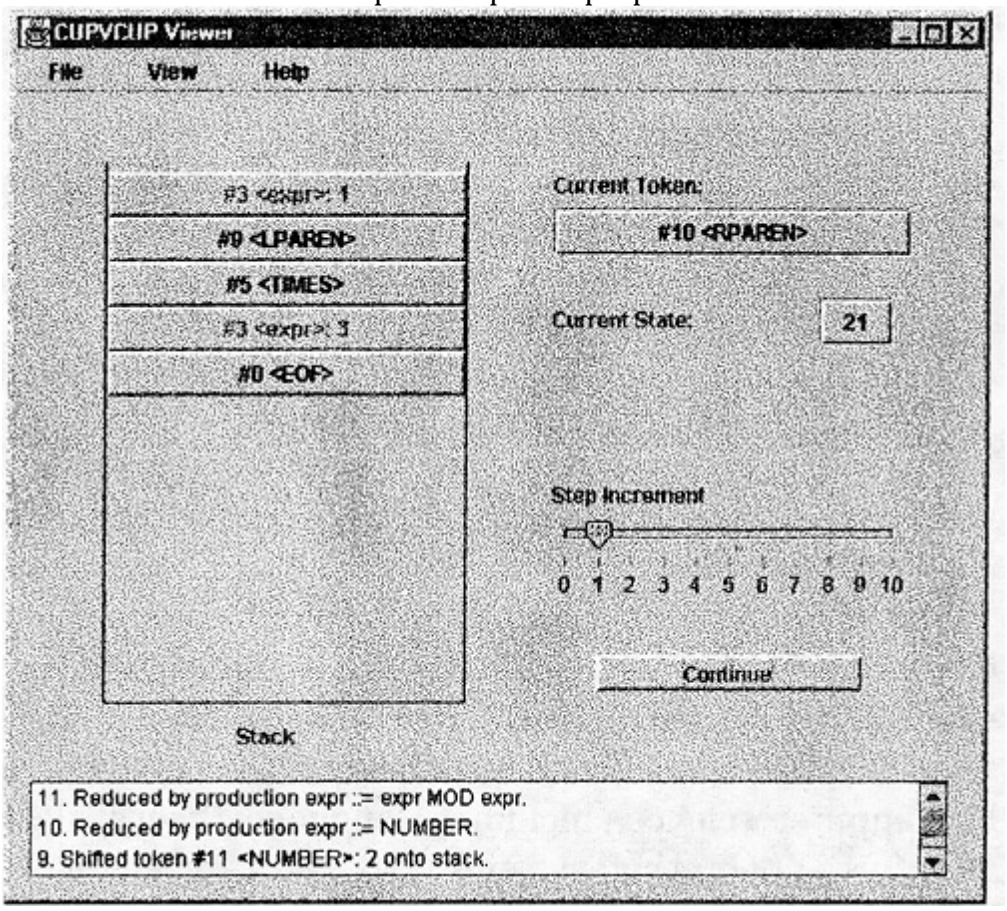


Figura 8. Captura de la herramienta SEFALAS (tomada de [77])

- BURGRAM [54], un simulador basado en visualizaciones orientado a mostrar el funcionamiento de distintos algoritmos implicados en el procesamiento de lenguajes, tales como los métodos de análisis sintáctico ascendentes y descendentes. La herramienta permite a los alumnos emular estos algoritmos con diferentes gramáticas descritas usando el generador ANTLR [124], y representa las diferentes estructuras de datos implicadas usando GRAPPA [62] (paquete de Java para el dibujo de grafos) o mediante tablas. Finalmente, la herramienta es capaz de registrar las acciones de los alumnos y exportarlas a formato PDF o HTML. De esta forma, los alumnos pueden adjuntar los ejercicios realizados con BURGRAM en sus apuntes.
- SEFALAS [77], un simulador orientado a la enseñanza de diferentes algoritmos de análisis léxico y sintáctico. La herramienta es capaz de emular algoritmos de análisis léxico con y sin anticipación, y algoritmos de análisis sintáctico ascendentes y descendentes. El simulador muestra las diferentes estructuras de datos implicadas, tales como el árbol sintáctico, el autómata finito determinista y la pila de análisis. El alumno deberá proporcionar al simulador las especificaciones léxicas, usando el lenguaje LEX [98], y las sintácticas, en formato YACC [98], para poder emular los algoritmos. La figura 8 muestra una captura de SEFALAS con toda la información anteriormente mencionada que es capaz de proporcionar.



- Figura 9. Captura de CUPV (extraída de [81])

- CUPV es una herramienta capaz de procesar especificaciones del generador CUP [72] y emular el comportamiento de la pila de análisis y el valor de cada una de sus posiciones. También es capaz de emular las diferentes

reducciones realizada por el analizador, pudiendo considerarse el simulador como un depurador visual para CUP. Además, en [81] se describe un estudio donde un grupo de alumnos de la asignatura de Procesadores de Lenguaje valoró la utilidad del simulador. El estudio muestra que el software es capaz de reducir el tiempo de depuración gracias a las visualizaciones con las que muestra el proceso de análisis, que resultan más intuitivas para los alumnos. La figura 9 presenta una captura de la herramienta.

Los simuladores analizados a lo largo de esta subsección se centran en proporcionar información adicional a los generadores de procesadores de lenguajes en los que se apoyan (CUP, ANTLR, etc.; véase el apartado 2.4 para una presentación más detallada de este tipo de generadores), para que los alumnos sean capaces de entender el proceso interno de análisis sintáctico soportado por este tipo de herramientas. Incluso, algunas de ellas son empleadas como depuradores altamente especializados por los alumnos durante el transcurso de la asignatura y para el desarrollo de los procesadores pedidos por los docentes como prácticas. Como contrapunto, estas herramientas se centran exclusivamente en ilustrar el funcionamiento interno de los procesadores implementados, dejando de lado los aspectos relativos a las especificaciones que preceden a dicha implementación.

2.2.2.3 Simuladores de otros aspectos del procesamiento

Aparte de simular los procesos de análisis, también es posible abordar la simulación de otros aspectos del procesamiento. De esta forma, este tipo de software muestra la construcción, actualización y consulta de, por ejemplo, la tabla de símbolos, el árbol de análisis sintáctico, o la ejecución del código objeto resultante. A continuación se describen tres ejemplos representativos de este tipo de sistemas:

- SOTA [165], un simulador que permite visualizar la tabla de símbolos de un procesador de lenguaje y simular las acciones que se realizan sobre ella. El software presenta simultáneamente el código fuente, una visualización de la tabla de símbolos y un histórico de las acciones que se realizan sobre ella. En [165] se describe un estudio realizado con alumnos donde se mide la valoración de estos y la eficacia educativa del software. En primer lugar, el estudio concluye que la eficacia educativa es similar a la conseguida utilizando el método de enseñanza tradicional para este concepto, pero la eficiencia educativa (entendida como la rapidez del alumno para entender el concepto) aumenta con respecto al método tradicional.
- *Tree-Viewer Library* [168], una biblioteca Java que ayuda a los alumnos a depurar sus procesadores, ya que es capaz de generar automáticamente visualizaciones de los árboles de análisis sintáctico. La figura 10 muestra una captura de la biblioteca en funcionamiento. De esta forma, los alumnos son capaces de convertir sus implementaciones en simuladores del proceso de construcción de árboles, lo que les ayuda a entender el procesamiento dirigido por la sintaxis. Su autor defiende que la herramienta es útil para docentes y alumnos, puesto que les permite depurar y comprender

visualmente, y de una forma bastante atractiva, cuáles pueden ser los errores del compilador que se está depurando.

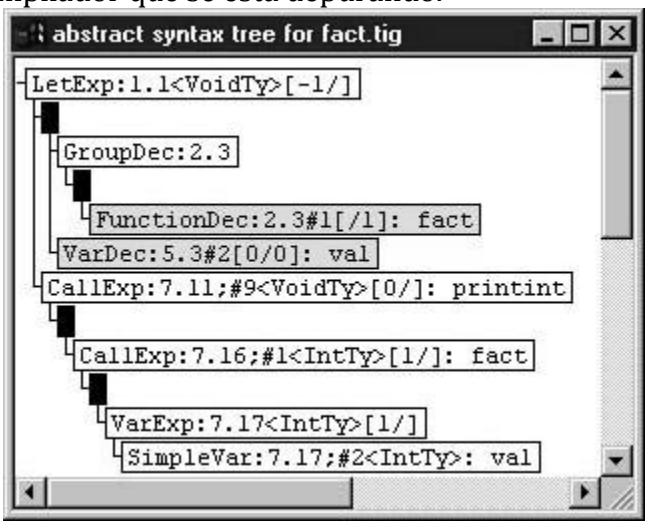


Figura 10. Captura de la visualización de un árbol sintáctico con Tree-Viewer Library (extraída de [168])

- *Annotating Debugger* [168], un depurador para una máquina virtual basada en pila que ejecuta el código máquina generado por un compilador. La herramienta muestra el funcionamiento de la máquina basada en pila de forma visual, junto con diferentes anotaciones incluidas en el código máquina por el docente. Estas anotaciones ayudan al alumno a entender cuál es el objetivo de esa instrucción máquina, con respecto al código del procesador del lenguaje. Aparte de su uso educativo, los autores proponen esta herramienta como un depurador visual para sus procesadores. La figura 11 presenta la ventana principal del depurador para la máquina virtual.

Las herramientas educativas estudiadas anteriormente, junto con otras similares (v.g., VAST para la visualización del proceso de construcción de árboles sintácticos [11] [12]) proporcionan a los alumnos una forma más atractiva de explorar diferentes estructuras de datos implicadas en el procesamiento del lenguaje y su función durante dicho proceso. Para ello, hacen uso de la simulación basada en visualizaciones donde se anima el flujo de los datos en cada una de estas estructuras. Su utilidad educativa ha sido demostrada en los diferentes estudios que se han descrito anteriormente, pero de nuevo estos sistemas están enfocados principalmente en aspectos relativos a la implementación, en lugar de en los aspectos previos de especificación en los que se basa dicha implementación.

2.2.3 Discusión

Todas las herramientas presentadas en las secciones anteriores están orientadas a mejorar la experiencia educativa dentro de distintas materias relacionadas con la Informática, y en concreto con las materias relacionadas con la construcción de procesadores de lenguajes. Estas herramientas hacen un uso exhaustivo del poder pedagógico de las visualizaciones para presentar la información e interacción de los conceptos [71] [73] [166], y se configuran como herramientas pedagógicas muy potentes para mejorar la enseñanza de

determinados conceptos relacionados con la Informática en general, y las materias relativas a los procesadores de lenguaje en particular, observándose multitud de experiencias significativas que permiten extraer buenas prácticas. En particular, en relación con las herramientas centradas en las materias de construcción de procesadores de lenguaje, se han presentado diferentes tipos en función de los conceptos que cubren. Independientemente de su ámbito, todas ellas proporcionan a los alumnos una forma más atractiva de interiorizar tanto conceptos teóricos de la materia como el funcionamiento de diferentes etapas durante el procesamiento del lenguaje.

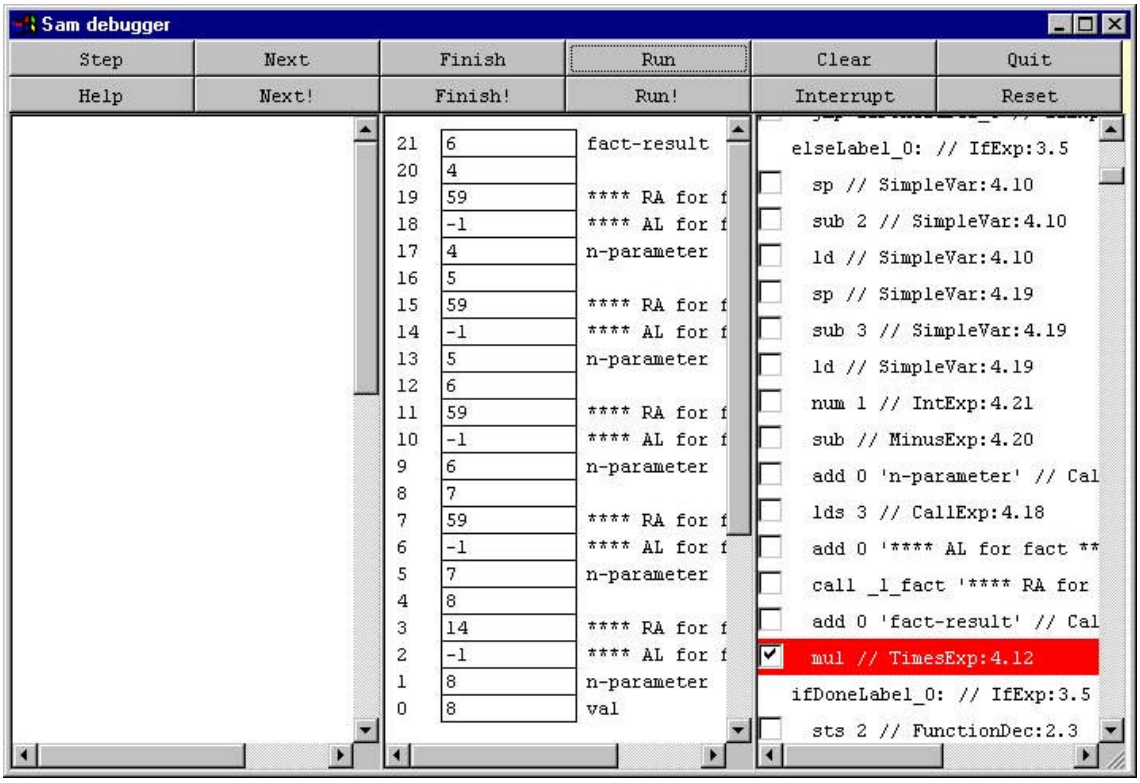


Figura 11. Captura de la visualización de la maquina basada en pila generada por Annotating Debugger (extraída de [168])

No obstante, y a pesar de las múltiples ventajas de los enfoques basados en simulación, es importante apuntar al hecho de que los simuladores relacionados con el curso de procesadores de lenguaje anteriormente descritos están fundamentalmente orientados a la enseñanza de procesos y conceptos relacionados con la implementación de los procesadores. Sin embargo, los aspectos de especificación no reciben apenas atención. Esto es evidente para los simuladores de los algoritmos de análisis y de otros aspectos del procesamiento, ya que se centran en la enseñanza de las estructuras y procesos internos de los procesadores. También es cierto para los simuladores de máquinas teóricas, ya que aunque estos pueden parecer más cercanos a la especificación, al estar orientados a la enseñanza de conceptos teóricos, estas máquinas son modelos abstractos de artefactos empleadas en la implementación de los procesadores de lenguajes. Por tanto, estos enfoques no contribuyen a mejorar la comprensión y el uso de los formalismos de especificación por parte de los estudiantes, sino a comprender mejor el funcionamiento de las implementaciones finales.

2.3 Juegos educativos

Esta tesis promueve también el uso de juegos serios como mecanismos para habilitar la comprensión de conceptos básicos sobre la especificación basada en gramáticas de atributos. De esta forma, en esta sección se revisan los aspectos más relevantes para esta tesis sobre videojuegos educativos, un tipo de artefacto que ha irrumpido con fuerza en el mundo del eLearning y el software educativo [13] [56] [57].

La idea de emplear juegos para enseñar no es algo nuevo: ya han sido empleados durante años para enseñar diferentes conceptos y ayudar a la adquisición de habilidades, sobre todo en la niñez [75]. De esta forma, buscar usos educativos de los videojuegos es algo completamente natural. En particular, los videojuegos pueden permitir al usuario aprender y experimentar con procesos y técnicas muy cercanas a los de la vida real, ya que pueden sumergirles en mundos virtuales diseñados para tales propósitos [156]. Los videojuegos educativos presentan cuatro características fundamentales que los posicionan como una herramienta pedagógica valiosa [56]:

- *Los videojuegos son divertidos*. Esta percepción no es sólo aplicable a la población joven, ya que un gran espectro de la población que ronda los 30 años son habituales consumidores de este tipo de ocio. Además, alrededor del 43% de los usuarios de este tipo de software son mujeres, lo que no supone una diferencia significativa por sexos. Teniendo en cuenta todo esto, podemos concluir que un amplio espectro de la población puede sentirse atraído a usar videojuegos.
- *Los videojuegos son inmersivos*. Estos programas sumergen a los usuarios en mundos virtuales con los que pueden interactuar. De esta forma, los diseñadores pueden crear mundos virtuales orientados a proporcionar una experiencia educativa a los usuarios.
- *Los videojuegos estimulan la cooperación y la competitividad*. Los videojuegos proporcionan herramientas para promocionar el aprendizaje colaborativo ya sea mediante el uso de juegos en red (por ejemplo, MMOG: *Massive Multiplayer Online Games*), o bien sin compañeros reales (mediante el uso de NPCs: *Non-Player Characters*). Además, determinados videojuegos pueden estimular la competitividad del usuario, lo que siempre resulta un estímulo muy poderoso.
- *Los videojuegos estimulan la creación de comunidades de usuarios*. Estas comunidades son una extensión de la actividad del videojuego, fuera del videojuego, donde los usuarios pueden compartir experiencias y aprender a ser mejores jugadores. Es importante apuntar que este fenómeno es difícil de encontrar en las escuelas convencionales, ya que la mayoría de los estudiantes no tratan temas escolares fuera de la escuela.

Teniendo en cuenta todo esto, los videojuegos educativos se posicionan, potencialmente, como un poderoso instrumento educativo. Actualmente, los videojuegos educativos se emplean en diferentes ámbitos y materias. En particular, en el ámbito de la Informática se han creado interesantes propuestas para facilitar el proceso de enseñanza y aprendizaje. Dichas propuestas se describen en las secciones que siguen: sección 2.3.1, y sección 2.3.2 en relación con el campo

particular de los procesadores de lenguaje. La sección 2.3.3 discute los aspectos más relevantes de cara al desarrollo de la tesis.

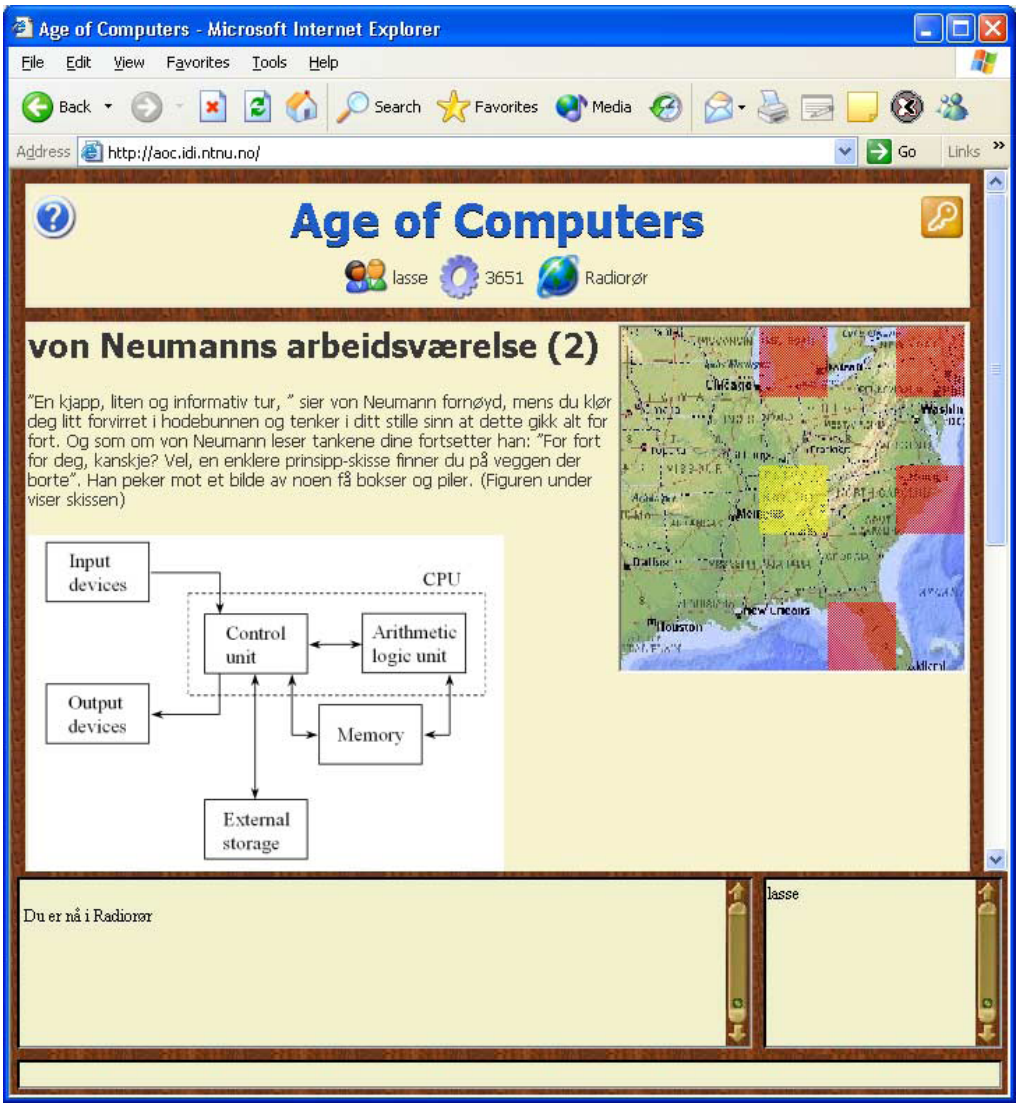


Figura 12. Captura del videojuego Age of Computers (tomada de [116])

2.3.1 Uso de juegos educativos en Informática

Esta sección analiza el uso de videojuegos educativos en relación con la enseñanza y el aprendizaje de materias propias de la Informática. Estos videojuegos están orientados a mejorar el aprendizaje de los alumnos en diferentes aspectos de las materias que componen la informática, tales como algoritmia, gestión de proyectos o arquitectura de computadores. Algunos ejemplos representativos son:

- *Age of Computers* [116]. Este es un videojuego educativo enfocado a mejorar el proceso de enseñanza–aprendizaje de los alumnos de primer curso, en materias sobre *Fundamentos de Computadores*. El videojuego, desarrollado por el *Instituto Noruego de Tecnología y Ciencia,* sitúa a los alumnos en diferentes épocas de la informática donde tendrán que resolver diferentes problemas. A medida que va resolviendo estos problemas, el alumno gana

puntos para avanzar a la siguiente época. Además, el videojuego proporciona un sistema de *chat* donde el usuario puede comunicarse con otros alumnos que estén jugando en su misma época, lo que estimula el aprendizaje colaborativo. La figura 12 ilustra cómo el videojuego presenta las preguntas al alumno. En [116] se presentan resultados positivos de satisfacción entre los alumnos que probaron el videojuego. Aun así, algunos alumnos consideran insuficiente el *chat* para resolver los problemas de forma colaborativa y solicitan nuevas formas de colaboración.

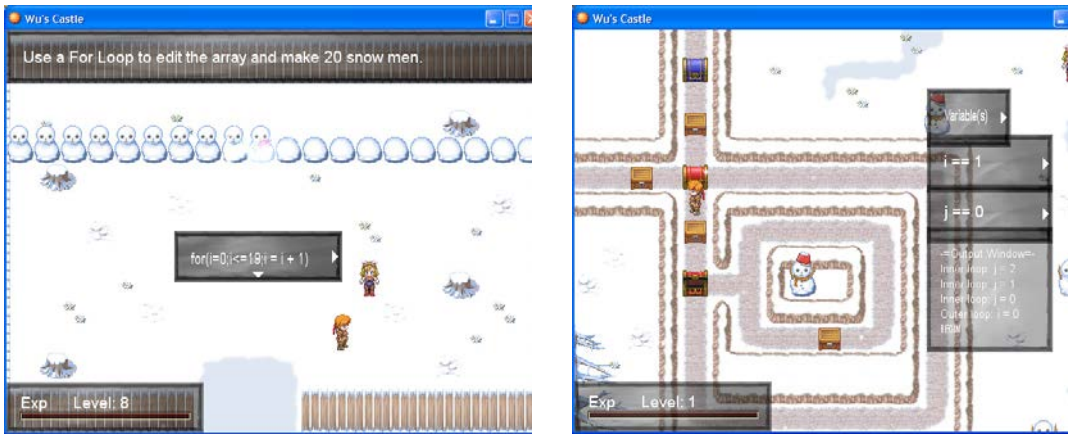


Figura 13. Capturas de las distintas modalidades de juego en Wu'sCastle (tomadas de [45])

- Wu's Castle [45], un videojuego educativo en dos dimensiones diseñado para enseñar el funcionamiento de los bucles a los alumnos de los primeros cursos. El videojuego presenta diferentes niveles que se basan en dos modos de juego. En el primero, el objetivo del juego es recorrer una fila, o varias, de muñecos de nieve usando un bucle *for*, que los alumnos tendrán que configurar y modificar. El otro modo presenta una especificación de un bucle y el usuario deberá simularlo controlando un personaje que recorrerá un laberinto. La figura 13 muestra los diferentes modos de juego descritos anteriormente. Para medir la satisfacción de los usuarios y la efectividad educativa del videojuego se realizó el estudio descrito en [46]. Los alumnos que participaron en el estudio fueron divididos en tres grupos: alumnos que únicamente utilizaron el videojuego, alumnos que no utilizaron el videojuego pero realizaron una batería de *tests* sobre la materia, y alumnos que utilizaron el videojuego y realizaron los *tests*. Tras analizar los resultados obtenidos, tanto en los *tests* como en la nota final del curso, se observó que los alumnos que utilizaron el videojuego obtuvieron una puntuación mayor que el resto de grupos.

- *Entrada/Salida en Nintendo® DS* [94]. Esta experiencia supone una estrategia educativa basada en el uso de videojuegos apoyados en la videoconsola Nintendo® DS para enseñar el funcionamiento de la entrada/salida en los computadores. La estrategia se divide en tres fases. Primero, los alumnos van respondiendo una serie de cuestiones teóricas, en parejas de dos, que contestarán de forma individual y simultáneamente. Si ambos aciertan, se llevan toda la puntuación. Si alguno falla, ninguno se llevará la puntuación entera. Esto último fomenta el aprendizaje

colaborativo, al depender la consecución de la tarea del compañero. Una vez superado el cuestionario, el videojuego les exige construir colaborativamente un autómata de entrada/salida. Finalmente, los alumnos responden de forma individual el primer cuestionario siguiendo el sistema Leitner [96].

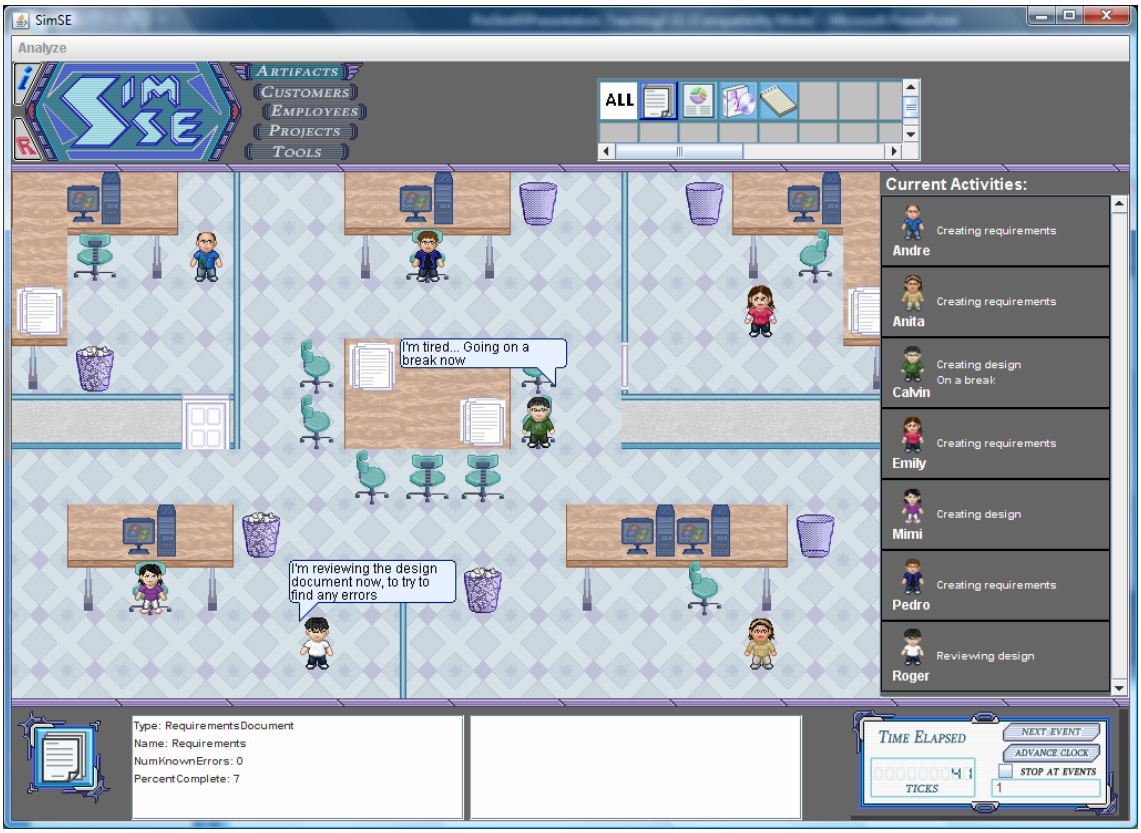


Figura 14. Captura del simulador SimSE (tomada de [117])

- SimSE [117]. Este es un juego de simulador que sitúa al usuario al frente de un grupo de desarrollo que debe dirigir para realizar los diferentes proyectos que el simulador proponga. A medida que el alumno avance en la simulación irá ganando puntos, en función de cómo aplique las buenas prácticas en la gestión de proyectos. Al terminar la simulación, el videojuego presenta un informe completo sobre la actividad del alumno, proporcionando una realimentación adecuada para aquellas malas decisiones que este haya podido tomar. La figura 14 muestra una simulación en curso de SimSE. A pesar de que en [117] se ha demostrado la eficacia educativa del simulador en diferentes universidades, el estudio también muestra que los alumnos tienen dificultades para empezar a utilizar la herramienta, necesitando ser instruidos en su manejo previamente.

Los videojuegos presentados anteriormente están orientados a la enseñanza de distintas materias relacionadas con la Informática. Todos ellos son capaces de enseñar a los alumnos, de forma atractiva, los correspondientes conceptos para los que fueron diseñados, utilizando metáforas que transformen adecuadamente dichos conceptos en videojuegos con valor educativo (por ejemplo, Wu's Castle).

Además, tienen la capacidad de proporcionar a los alumnos, de forma inmediata, información sobre los fallos (o aciertos) que estos puedan cometer, para guiar su aprendizaje. Por otro lado, algunos de los ejemplos analizados cuentan con características que potencian el aprendizaje colaborativo (por ejemplo, *Age of Empires*) o que permiten a los docentes evaluar el avance de los alumnos (por ejemplo, SimSE). A modo de conclusión, estos ejemplos sientan precedentes del uso de videojuegos educativos para la enseñanza de conceptos relacionados con la Informática, e ilustran las bondades y características de este tipo de software educativo.

2.3.2 Uso de juegos educativos en Procesadores de Lenguaje

En cuanto a las materias relacionadas con la construcción de procesadores de lenguaje, en lo referente a videojuegos educativos no existe una gran variedad de estos que cubran específicamente los conceptos enseñados durante la materia. Quizá una de las excepciones sea JV²M, una herramienta educativa basada en videojuegos que se orienta a la enseñanza del funcionamiento de la máquina virtual de Java y la traducción de los programas escritos en Java en *bytecode* [58]. El objetivo principal es que los alumnos comprendan la traducción de los programas Java en *bytecode* mediante la comprensión de la ejecución interna de estos programas. El videojuego presenta al usuario una ciudad donde cada edificio se identifica con una parte de la máquina virtual, y el usuario deberá interactuar con estos edificios (y otros objetos) para ejecutar el programa Java propuesto por el videojuego. La figura 15 muestra diferentes capturas del videojuego educativo.

Este sistema educativo presenta dos características muy interesantes:

- En primer lugar, el videojuego cuenta con la inclusión de un *tutor inteligente* [180], representado como un acompañante del avatar que representa al usuario que puede proporcionar realimentación adicional o incluso resolver el problema propuesto.
- Por otro lado, la arquitectura interna del sistema ha sido concebida para separar claramente los contenidos a enseñar de la forma de presentarlos. Así, JV²M es capaz de configurar los niveles de juego a partir de descripciones de dichos problemas [58].

Figura 15. Capturas de JV²M (tomadas de [58])

Es importante destacar que este videojuego educativo ha obtenido buenos resultados entre los estudiantes [58]. También es importante resaltar la falta de videojuegos educativos orientados a la enseñanza de materias relacionadas con el diseño e implementación de lenguajes informáticos.

2.3.3 Discusión

Durante esta sección se han descrito las diferentes características y cualidades de los videojuegos que los convierten en una poderosa herramienta educativa, principalmente debido al atractivo que poseen y que invita a los alumnos a utilizarlos, y a aprender con ellos. Este hecho justifica también su uso en diversas materias relacionadas con la Informática.

No obstante, cabe destacar el reducido número de videojuegos educativos orientados específicamente a la enseñanza - aprendizaje de las materias relacionadas con la construcción de procesadores de lenguajes. Esto probablemente sea debido a que los conceptos y técnicas involucrados en las materias afines al diseño e implementación de lenguajes informáticos no se prestan con facilidad a ser enseñados usando juegos serios debido a su naturaleza abstracta. En última instancia, puede resultar difícil encontrar metáforas adecuadas para estas materias, y que a la vez resulten entretenidas y atractivas para los alumnos. Estas metáforas determinan cómo los conceptos o procesos que se van a tratar en un juego serio van a representarse a través de una mecánica de juego atractiva y adecuada a los alumnos, y sus necesidades educativas.

Como excepción a la carencia de juegos serios aplicados al campo de la enseñanza – aprendizaje de los procesadores de lenguaje cabe destacar el ejemplo analizado en la sección anterior, JV²M, que ilustra a los alumnos el funcionamiento de la traducción de código Java a *bytecode* de la máquina virtual de Java. Mientras que JV²M ilustra claramente la factibilidad de utilizar juegos serios en relación con este tipo de materias, dicho sistema se centra fundamentalmente en los aspectos de implementación del proceso de traducción y de la ejecución de los programas objeto en máquinas virtuales, sin abordar los aspectos de especificación previos.

2.4 Herramientas para el desarrollo de procesadores de lenguaje

Tal y como se ha mencionado en el apartado 2.1, los cursos típicos en las materias relacionadas con el diseño e implementación de lenguajes se dividen en dos partes fundamentales: por un lado, la enseñanza de los conceptos teóricos y técnicas que componen la materia, y por otro lado, la aplicación de estos para el desarrollo de un procesador. En relación con la parte práctica de desarrollo, existen dos posibilidades, ambas ampliamente utilizadas:

- Realizar el desarrollo directamente en un lenguaje de programación de propósito general. Esta alternativa es viable para lenguajes pequeños y para técnicas de implementación sencillas (por ejemplo, traductores predictivo – recursivos [5]). Sin embargo, para lenguajes más complejos o técnicas de implementación más elaboradas, el enfoque deja de ser practicable.

- Utilizar herramientas software específicas para desarrollar procesadores de lenguajes. Dichas herramientas operan generalmente a partir de especificaciones de alto nivel de distintos aspectos del lenguaje a procesar y de su procesador. Aparte de facilitar el desarrollo, dichas herramientas son especialmente atractivas desde el punto de vista educativo, al permitir poner énfasis sobre los aspectos de especificación en lugar de sobre los aspectos de implementación.

De esta forma, en esta tesis las herramientas de desarrollo de procesadores de lenguaje juegan un papel central, al ser capaces de aceptar como entrada distintos tipos de especificaciones de procesadores de lenguaje, enfatizando los aspectos relativos a la especificación. En este apartado se discuten los aspectos de estas herramientas más relevantes para el desarrollo de la tesis. La sección 2.4.1 discute las herramientas de generación de procesadores basadas en el formalismo de las gramáticas de atributos. La sección 2.4.2 discute las herramientas más convencionales, basadas en esquemas de traducción. La sección 2.4.3 discute cómo utilizar dichas herramientas basadas en esquemas de traducción para implementar especificaciones basadas en gramáticas de atributos. La sección 2.4.4 incide en el uso educativo de las herramientas de procesadores de lenguaje. La sección 2.4.5 discute, por último, los aspectos más relevantes de los diferentes elementos tratados en el apartado.

2.4.1 Herramientas basadas en gramáticas de atributos

Las herramientas basadas en gramáticas de atributos aceptan como entrada especificaciones orientadas a gramáticas de atributos y producen como resultado los procesadores de lenguaje especificados. A continuación se revisa el formalismo básico de las gramáticas de atributos en el que se basan dichas herramientas, así como las características más relevantes de las mismas.

2.4.1.1 Gramáticas de atributos

El formalismo de las gramáticas de atributos fue propuesto por Donald E. Knuth a finales de los años sesenta del siglo pasado para añadir aspectos semánticos a las gramáticas incontextuales [89]. Las gramáticas de atributos describen un estilo de procesamiento dirigido por la sintaxis, ya que el procesamiento de cada frase es dirigido por su estructura sintáctica, y conducido por las dependencias, debido a que los cálculos semánticos están dirigidos por las dependencias implícitas en las computaciones involucradas. La figura 16 muestra un ejemplo de una especificación en forma de gramática de atributos que caracteriza la semántica de un lenguaje para describir expresiones aritméticas simples seguidas de una sección de declaración de constantes. Con el fin de simplificar el ejemplo, y hacer más claras las explicaciones, el lenguaje sólo contempla el operador *suma*. De acuerdo con dicha especificación, durante el procesamiento de las expresiones los valores de las constantes son almacenados en una estructura compuesta por un par *nombre-valor*. La evaluación de las expresiones en sí aprovecha su estructura sintáctica, así como la información contextual almacenada en la citada estructura.

```
Sent ::= Exp where Decs
   Exp.env↓ = Decs.env↑
   Sent.val↑ = Exp.val↑
Exp ::= Exp + Opnd
   Exp1.env↓ = Exp0.env↓
   Opnd.env↓ = Exp0.env↓
   Exp0.val↑ =Exp1.val↑+Opnd.val↑
Exp ::= Opnd
   Opnd.env↓ = Exp.env↓
   Exp.val↑ = Opnd.val↑
Opnd ::= num
   Opnd.val↑ = toNum(num.lex↑)
Opnd ::= var
   Opnd.val↑ = valOf(var.lex↑, Opnd.env↓)
Opnd ::= (Exp)
   Exp.env↓ = Opnd.env↓
   Opnd.val↑ = Exp.val↑
Decs ::= Decs, Dec
   Decs0.env↑ = extendWith(Dec.env↑,
         Decs1.env↑)
Decs ::= Dec
   Decs.env↑ = Dec.env↑
Dec ::= var = num
   Dec.env↑ = {(var.lex↑,toNum(num.lex↑))}
```

Figura 16. Ejemplo de una gramática de atributos

De esta forma, las gramáticas de atributos extienden el formalismo de las *gramáticas incontextuales* para dotarlas de poder semántico mediante *atributos semánticos* y *ecuaciones semánticas*. Las gramáticas incontextuales son ampliamente utilizadas para describir la sintaxis de los lenguajes de programación. Este formalismo se compone de los siguientes elementos:

- Las construcciones sintácticas se representan por *símbolos sintácticos*: los símbolos *no terminales* se corresponden con estructuras compuestas y los símbolos *terminales* con las construcciones del lenguaje más simples. Por ejemplo, en la figura 16 los símbolos `Sent`, `Exp` y `Decs` son símbolos no terminales, y **`where`**, **`num`**, **`var`** y **`=`** se corresponden con símbolos terminales.
- Para cada uno de los símbolos no terminales habrá como mínimo una *regla sintáctica* (o *producción*) definiendo su estructura sintáctica. Cada regla está compuesta por su parte izquierda (LHS o cabeza de la regla) que se corresponde con el símbolo no terminal a definir, y la parte derecha (RHS o cuerpo de la regla) que se compone de una secuencia de símbolos del lenguaje (terminales o no). Por ejemplo, la regla `Sent ::= Exp` **`where`** `Decs` establece que una sentencia `Sent` está compuesta por: una expresión `Exp`, seguida de la palabra reservada **`where`** y concluyendo con un bloque de declaración de constantes `Decs`.
- Además, existe un no terminal llamado *axioma* (o *símbolo inicial* de la gramática) que representa la estructura de mayor nivel dentro del lenguaje definido. El símbolo `Sent` de la figura 16 es el símbolo inicial de la gramática.

En las gramáticas incontextuales, las reglas sintácticas imponen una estructura jerárquica sobre las sentencias del lenguaje que definen. Esta estructura se puede representar de forma gráfica mediante un árbol, conocido como *árbol sintáctico*. En este árbol, los nodos internos se corresponden con símbolos no terminales de la gramática, y las hojas con símbolos terminales. Cada nodo padre, junto con sus nodos hijo, se corresponden con la aplicación de una regla sintáctica. Finalmente, la raíz del árbol se corresponde con el axioma de la gramática. La figura 17a muestra una sentencia de ejemplo del lenguaje descrito en la figura 16, y la figura 17b representa el árbol sintáctico producto de procesar la sentencia. Obsérvese

cómo el árbol hace explícita la estructura de la sentencia. Usando esta estructura como hilo conductor se realizan los diferentes procesamientos de la sentencia.

(a) `x+y+5 where x=5,y=6`

(b)

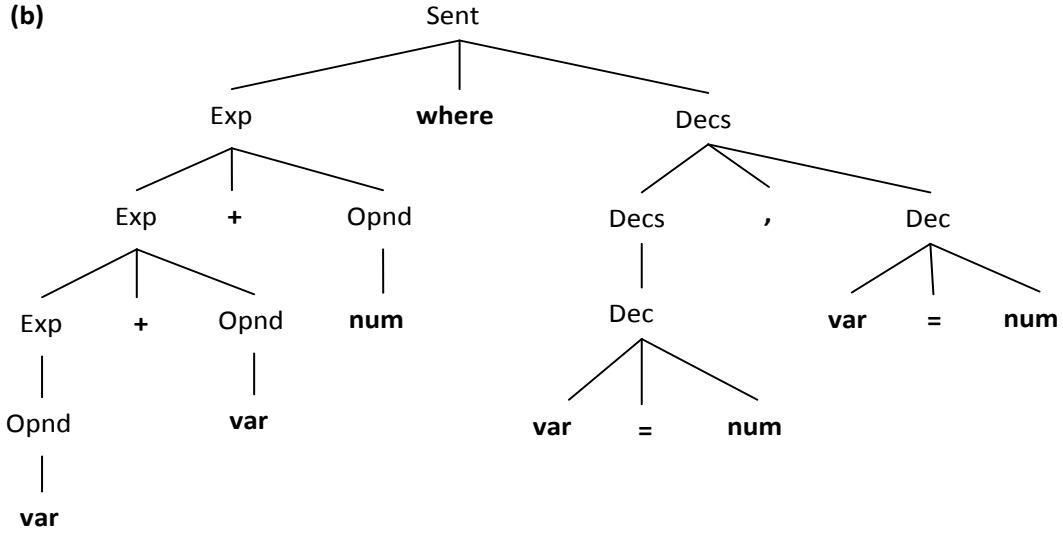


Figura 17. (a) Una sentencia del lenguaje definido en la figura 16, (b) Árbol de análisis sintáctico para la sentencia descrita en (a).

Como se ha indicado anteriormente, las gramáticas de atributos añaden un conjunto de *atributos semánticos* a los símbolos del lenguaje. Estos atributos almacenarán valores en los correspondientes nodos del árbol de análisis sintáctico. Existen dos tipos de atributos:

- *Atributos sintetizados*: sus valores son calculados a partir de los valores de atributos sintetizados que se encuentran en los nodos hijo, respecto al nodo en el que se encuentra el atributo cuyo valor se va a calcular, y los valores de los atributos heredados del propio nodo. Así, el valor de los atributos sintetizados representan (parte de) el significado del símbolo del lenguaje al que está asociado el atributo. En la gramática de la figura 16, los atributos sintetizados están acompañados del símbolo ↑. Por ejemplo, el atributo `val`↑ en la figura 16 almacenará el valor numérico de la expresión procesada. Finalmente, los atributos sintetizados asociados a los símbolos terminales se conocen como *atributos léxicos*, y su valor se fija durante el análisis léxico de la sentencia. Un ejemplo de atributo léxico es `lex`↑ en la figura 16, que almacena el lexema del símbolo terminal (el número concreto, la variable concreta, etc.).
- *Atributos heredados*: sus valores se calculan a partir de los valores de los atributos heredados del nodo padre y/o de los atributos sintetizados de sus nodos hermanos. Los atributos de esta clase almacenan información contextual adicional necesaria para determinar el valor de los atributos asociados con el nodo al que pertenecen. En la gramática de la figura 16, estos atributos heredados se acompañan del símbolo ↓

- Por ejemplo, `env↓` en la figura 16 mantiene las constantes declaradas para ser usadas durante el cálculo del valor de la expresión numérica.

Este formalismo añade, además, un conjunto de *ecuaciones semánticas* a cada una de las reglas sintácticas. Estas ecuaciones indican cómo se calculan los valores de los atributos de una regla sintáctica: los atributos sintetizados de la parte izquierda de la regla, y los atributos heredados de la parte derecha de la regla. De forma más precisa:

- Habrá una ecuación semántica por cada atributo sintetizado de la parte izquierda de la regla, y una por cada atributo heredado de la parte derecha de la regla.
- Cada ecuación utiliza *funciones semánticas* aplicadas sobre otros atributos de la regla. Asumiremos que, dentro de la computación expresada en cada ecuación, sólo emplearemos atributos heredados de la parte izquierda de la regla y atributos sintetizados de la parte derecha de la regla (es decir, consideraremos las gramáticas de atributos en la forma normal de *Bochmann* [25]).

Por ejemplo, la ecuación semántica $Exp_0.val\uparrow = Exp_1.val\uparrow + Opnd.val\uparrow$ de la regla sintáctica `Exp ::= Exp + Opnd` en la gramática descrita en la figura 16 establece que el valor de esa expresión ($Exp_0.val\uparrow$) se calcula como la suma del valor de ambos operandos ($Exp_1.val\uparrow$ y $Opnd.val\uparrow$).

Las gramáticas de atributos posibilitan la evaluación semántica en los árboles de análisis *decorados* (es decir, árboles de análisis donde los nodos están acompañados de atributos semánticos). La evaluación semántica está dirigida por las dependencias entre atributos, que imponen restricciones en el orden en que se deben calcular los atributos (es decir, para calcular el valor de un atributo, es necesario tener calculados los valores de los atributos de los que depende). Aparte de esta restricción básica, el orden de evaluación no importa. En consecuencia, las gramáticas de atributos son consideradas como un formalismo de alto nivel, ya que es posible especificar tareas de procesamiento de lenguaje centrándose en el sentido semántico de las estructuras sintácticas, sin tener en cuenta los detalles de implementación a bajo nivel, como el orden exacto de evaluación de los atributos. Además, el formalismo es altamente modular: esto facilita la adición de nuevos atributos y ecuaciones semánticas sin afectar a las ya existentes, ya que las dependencias entre los atributos serán las responsables de reorganizar automáticamente el orden de evaluación.

Una manera conveniente de describir dependencias entre atributos en un árbol de análisis sintáctico es mediante un *grafo de dependencias*. Los nodos de este grafo son los atributos asociados a los símbolos del árbol. Cada arco describe qué atributo debe ser usado para calcular el valor del atributo objetivo. La figura 18 muestra el grafo de dependencias para la sentencia de la figura 17a.

Una gramática de atributos es *no circular* cuando no es posible encontrar una instancia de atributo en un árbol sintáctico que dependa de sí misma, directa o indirectamente. Por el contrario, las gramáticas que no cumplen esta condición se conocen como gramáticas de atributos *circulares*. Aunque la evaluación semántica puede adaptarse para realizarse sobre gramáticas circulares (ver por ejemplo [78]), las gramáticas no circulares son las que se utilizan en la práctica para definir

la traducción de los lenguajes. La evaluación semántica en estas gramáticas puede definirse como [5]:

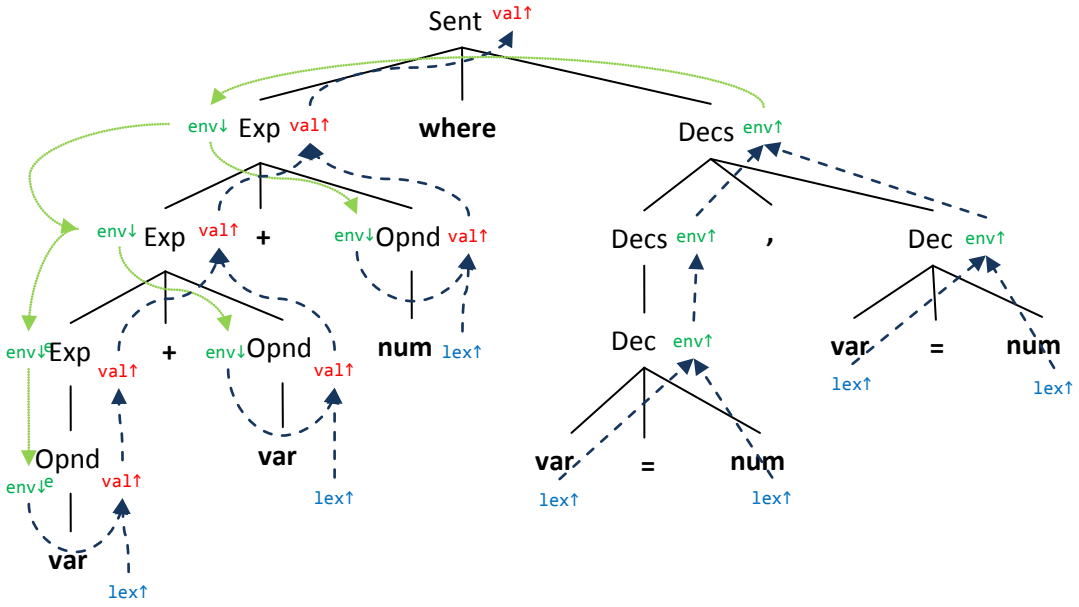


Figura 18. Árbol sintáctico decorado y su grafo de dependencias para la sentencia de la figura 2a

- Primero, se encuentra un orden topológico de los nodos en el grafo de dependencias para la sentencia procesada (ya que la gramática es no circular, el grafo de dependencias será acíclico). En este orden, cada instancia de atributo será precedida por todas las instancias de las que depende.
- Después, se evalúan las instancias de los atributos de acuerdo a este orden.

Sin embargo, este es un modelo de evaluación conceptual. En la práctica, la evaluación semántica se puede llevar a cabo utilizando diferentes estrategias cuya única restricción viene impuesta por las dependencias entre atributos. Asimismo, una estrategia de evaluación particular puede no requerir la construcción explicita del árbol de análisis sintáctico. De hecho, para diferentes subclases de las gramáticas de atributos (como la de las *gramáticas s-atribuidas*, que involucran sólo a atributos sintetizados, y algunas clases de *gramáticas l-atribuidas*, en donde los atributos heredados de un nodo solo dependen de atributos heredados localizados en su padre o sintetizados en los hermanos que lo preceden), es posible crear implementaciones que evalúan los atributos *al vuelo* durante el procesamiento de la sentencia, sin construir explícitamente el árbol sintáctico. Obsérvese que la gramática de la figura 16 no es s-atribuida (ya que necesita propagar un atributo heredado con el contexto necesario), ni l-atribuida (porque las declaraciones están situadas después de la expresión, y los valores de las constantes se necesitan para calcular el valor de la expresión).

2.4.1.2 Herramientas

Partiendo del formalismo básico, tal y como se explica en [120], la comunidad de lenguajes informáticos ha propuesto una gran cantidad de herramientas para

implementar procesadores de lenguajes a partir de una especificación en forma de gramáticas de atributos. Tales herramientas van desde sistemas clásicos como GAG [82], FNC-2 [80], ELI [63] o Elegant [16] hasta sistemas más modernos como LISA [70] [108] [118], Silver [181] o JastAdd [101]. Como ya se ha indicado, estas herramientas toman como entrada una especificación basada en el formalismo de las gramáticas de atributos y generan, como salida, un procesador de lenguaje operativo para el lenguaje definido. Así mismo, muchas de estas herramientas soportan metalenguajes que incluyen diferentes extensiones al formalismo básico de las gramáticas de atributos (e.g., módulos [84], genéricos [148], orden superior [172], objetos [68] y orientación a aspectos [128] [129]), lo que posibilitan la creación y mantenimiento de especificaciones más complejas.

El código generado por estas herramientas está usualmente muy optimizado, siguiendo un enfoque estático en sus estrategias de evaluación y almacenamiento, que son determinadas como resultado de un análisis estático de la especificación de entrada [1]. En definitiva, ese código generado no es susceptible de ser inspeccionado o modificado por personas.

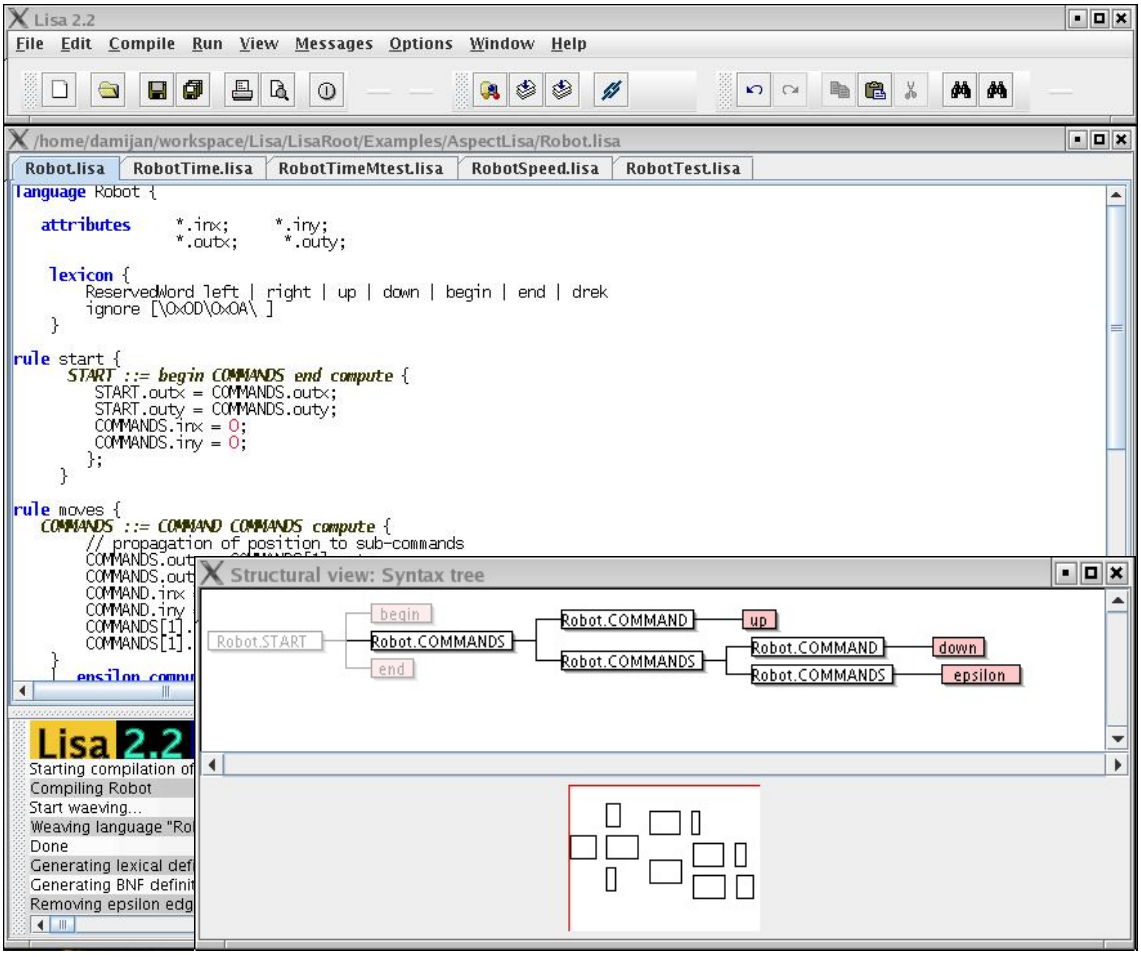


Figura 19. Captura del entorno de desarrollo LISA.

2.4.1.2.1 LISA

Para ilustrar la funcionalidad de las herramientas basadas en gramáticas de atributos, se describe con más detalle las características de LISA. LISA es un

entorno para el desarrollo de procesadores de lenguaje basados en gramáticas de atributos que proporciona a los usuarios editores textuales y visuales sofisticados que les orientan durante el desarrollo. Además, las diferentes partes que componen los procesadores generados (léxica, sintáctica y semántica) pueden ser creadas para que trabajen de forma separada e independiente. Por otro lado, la herramienta posibilita la reusabilidad de las especificaciones previamente definidas ofreciendo características como la modularidad y la herencia entre gramáticas. Finalmente, LISA es capaz de proporcionar visualizaciones y animaciones de las diferentes estructuras de datos implicadas en el proceso de análisis, como el árbol sintáctico, el autómata del reconocedor léxico y un depurador del proceso de evaluación semántica. La figura 19 muestra una captura del entorno LISA, donde se ilustra el editor de gramáticas de atributos y la representación del árbol sintáctico que ofrece.

La figura 20 muestra un ejemplo de especificación LISA para un lenguaje para la definición de constantes numéricas. Se puede observar que el léxico queda definido previamente a las reglas sintácticas y que su sintaxis es similar a la empleada en los libros de texto tradicionales para la enseñanza de las gramáticas de atributos.

```
language NumConst {
   lexicon {
       Identifier [a-zA-z]+
   }

   attributes java.util.Hashtable<String, Integer> *.table;

   rule Decs {
       DECS ::= DECS , DEC compute {
           DECS[1].table.putAll(DEC.table);
           DECS[0].table = DECS[1].table;
       };
       DECS ::= DEC compute {
           DECS.table = DEC.table;
       };
   }

   rule Dec {
        DEC ::= #Identifier #Number compute{
          DEC.table = new _Hashtable();
          DEC.table.put(#Identifier.value(),
          Integer.valueOf(#Number.value());
        };
   }

}
```

Figura 20. Especificación LISA de un lenguaje de definición de constantes numéricas

2.4.2 Herramientas basadas en esquemas de traducción

Aparte de las gramáticas de atributos, los *esquemas de traducción* son también formalismos ampliamente utilizados como mecanismo de especificación en gran cantidad de herramientas de generación de procesadores de lenguaje. Las siguientes subsecciones analizan este tipo de formalismos y de herramientas asociadas.

2.4.2.1 Esquemas de traducción

Los esquemas de traducción constituyen otro formalismo que extiende las gramáticas incontextuales con el fin de especificar el procesamiento dirigido por la sintaxis [5]. Al contrario que las gramáticas de atributos, que tienen un carácter más declarativo, los esquemas de traducción tratan con los aspectos procedimentales del procesamiento. Para este propósito:

(a)

```
Num ::= Num Bit {$$ := $1*2+$2}
Num ::= Bit {$$ := $1}
Bit ::= 0 {$$ := 0}
Bit ::= 1 {$$ := 1}
```

(b)

```
N(↑v) ::= Num(0,v)
Num(↓cv, ↑v) ::= Bit(vb) RNum(vb, v)
RNum(↓cv, ↑v) ::= Bit(vb) RNum(cv*2+vb, v)
RNum(↓cv, ↑v) ::= {v:= cv}
Bit(↑v) ::= 0 {v := 0}
Bit(↑v) ::= 1 {v := 1}
```

Figura 21. Ejemplos de esquemas de traducción bottom-up (a) y top-down (b)

- Los esquemas de traducción especifican un orden de visita de los nodos del árbol sintáctico. Los órdenes de visita más frecuentemente utilizados son *left-to-right bottom-up* y *top-down*. En ambos los nodos son visitados de izquierda a derecha, pero en el orden *bottom-up* los nodos son visitados en post-orden, y en el orden *top-down* son visitados en pre-orden. Además, en el orden *bottom-up* la visita de cada nodo sólo tiene un *punto significativo*, situado al finalizar la visita de todos sus nodos hijo. Por otro lado, el orden *top-down* ofrece diferentes puntos significativos: (i) cuando se empieza a visitar el nodo, (ii) después de visitar un nodo hijo y antes de empezar a visitar el siguiente, y (iii) cuando se termina de visitar el propio nodo.
- Los esquemas de traducción también especifican la forma en que se almacenará la información semántica. Por ejemplo, los modelos típicos de ejecución para la traducción *bottom-up* utilizan pilas semánticas para almacenar la información, mientras los modelos de ejecución más usuales para la traducción *top-down* asumen implementaciones basadas en subprogramas recursivos, usando sus parámetros y la pila de ejecución como mecanismos de almacenamiento semántico. Además, ambos tipos pueden usar variables globales para facilitar la definición de la traducción.
- Estos artefactos conciben las reglas sintácticas como planes de visita. Para este propósito, el formalismo introduce un mecanismo de referencias semánticas para consultar y actualizar la información semántica, así como permite introducir fragmentos de código (acciones semánticas) en los puntos significativos del recorrido. Las acciones semánticas definidas serán

ejecutadas cuando cada uno de estos puntos significativos sea visitado durante el proceso de análisis. Teniendo en cuenta los puntos significativos que ofrecen cada uno de los tipos de traducción, la traducción *top-down* proporciona una forma más natural para definir el procesador, siendo especialmente útil para gestionar la información semántica heredada.

```
PARSER_BEGIN(MiParser)
classMiParser {

}
PARSER_END(MiParser)

SKIP:{<["\t"," ", "\r","\b","\n"]>}
TOKEN : { <ID: <LETTER> (<LETTER>)*> | <#LETTER: [ "a"-"z", "_", "A"-"Z"] > }
TOKEN : { <NUM : "0" | ("1"-"9" ("0"-"9")*)>}

Map<String, Integer>Decs() :
{ Map<String, Integer>dec;
  Map<String, Integer>decs;}
{
  {} {return new HashMap<String, Integer>();} |
      dec = Dec() "," decs = Decs() {
            decs.addAll(dec);
            return decs;
      }
}

Map<String, Integer Dec() :
{ Map<String, Integer>dec;
Token name;
Token value;}
{
name = <ID> value = <NUM> {
           dec = new HashMap<String, Integer>();
           dec.put(name.image, Integer.parseInt(value.image));
           return dec;
      }
}
```

Figura 22. Especificación JavaCC de un lenguaje de definición de constantes numéricas.

La figura 21a muestra un ejemplo de esquema de traducción *bottom-up*. El lenguaje descrito es el clásico lenguaje de los números binarios propuesto por Knuth en [89] para ilustrar conceptos básicos sobre el formalismo de las gramáticas de atributos. El procesamiento consiste en calcular el valor decimal del número binario representado por la sentencia. Se ha usado una notación similar a la empleada por la herramienta YACC (una herramienta para la generación de traductores a partir de esquemas de traducción *bottom-up* [154]) para referenciar los valores semánticos de la pila de análisis. La figura 21b muestra un ejemplo de

esquema de traducción *top-down*, predictivo recursivo para el mismo lenguaje basado en los números binarios. La gramática subyacente ha sido transformada a LL(1), y las acciones semánticas han sido modificadas en consecuencia. Por lo tanto, la especificación resultante puede ser implementada en cualquier herramienta de generación de analizadores de este tipo. Al igual que en el caso anterior, no se ha seguido una notación concreta para una herramienta de generación, sino que la notación utilizada es asimilable a la de las herramientas más habituales que soportan este tipo de esquemas de traducción. Además, se ha anotado los parámetros de entrada con ↓ y los de salida con ↑.

2.4.2.2 Herramientas

Las herramientas de construcción de procesadores de lenguaje basadas en esquemas de traducción se adhieren normalmente a un modelo específico de esquema de traducción. En particular:

- Las herramientas de generación de traductores por desplazamiento-reducción LR utilizan esquemas de traducción *bottom-up* como entrada. Por tanto, los procesadores resultantes usan una pila para almacenar los valores semánticos. Además pueden utilizar variables globales para almacenar información semántica adicional. Este tipo de herramientas de generación está restringido normalmente a esquemas formulados sobre gramáticas incontextuales del tipo LR (normalmente, LALR(1) [5], aunque existen herramientas de este tipo que aceptan otros tipos más generales de gramáticas [29] [106]).
- Los esquemas de traducción *top-down* se utilizan como formalismo de especificación para herramientas de generación de traductores descendentes predictivos. Ya que muchas de estas herramientas generan analizadores recursivos descendentes, la información semántica se gestiona como parámetros de entrada y salida de los subprogramas generados, así como en variables locales. Al igual que en los *bottom-up*, los procesadores generados no construyen explícitamente el árbol sintáctico. Estas herramientas imponen unas fuertes restricciones a las gramáticas: gramáticas LL. Aunque las herramientas modernas de generación, como ANTLR, proporcionan extensiones a las gramáticas LL(k) (en particular, soportan los métodos de análisis LL(*), que proporcionan predicción ilimitada [125] [124]), no son capaces de manejar adecuadamente la recursión a izquierdas, por ejemplo. Aun así, estas herramientas proporcionan una forma más natural de gestionar la información heredada.

A continuación se analizan ambos tipos de herramientas.

2.4.2.2.1 Herramientas de generación de traductores descendentes

Algunos ejemplos de herramientas que generan traductores descendentes son JavaCC [91], ANTLR [124] y COCO/R [114]. Para ilustrar las diferentes características de estas herramientas se revisan JavaCC y ANTLR:

```
grammar MiParser;

@parser::header {
    importjava.util.*;
}

decs returns[Map<String, Integer>ts]:
{ $ts = new HashMap<String, Integer>();}
  dec {$ts.put($dec.name, $dec.val);}
  ("," dec {$ts.put($dec.name, $dec.val);})*
;

dec returns [String name, intval]:
 ID NUM {$name = $ID.text; $val = Integer.valOf($NUM.text)}
;

ID : ('a'..'z'|'A'..'Z'|'_')+;
NUM : '0' | ('1'..'9')('0'..'9')*;
WS  :   [ \t\n\r]+ -> skip ;
```

Figura 23. Especificación ANTLR de un lenguaje de definición de constantes numéricas.

- JavaCC [91] es un generador de traductores *top-down* basado en gramáticas LL(k). La herramienta es capaz de generar implementaciones en Java tras analizar especificaciones gramaticales, en formato EBNF, y léxicas, descritas mediante expresiones regulares. Las acciones semánticas se describen utilizando Java. La figura 22 muestra una codificación JavaCC del lenguaje para la definición de constantes numéricas introducido en el punto 2.4.1.2.1. Se observa que la especificación léxica y la gramatical están incluidas en el mismo fichero. Al estar basado en gramáticas LL(k), los desarrolladores pueden fijar cuántos símbolos preanalizará el *parser* generado, e incluso ofrece una versión donde este número de símbolos de preanálisis es adaptativo. Adicionalmente, JavaCC incluye dos herramientas: JJTree, para dotar a los procesadores creados con un constructor de árboles sintácticos, y JJDoc, que es capaz de generar automáticamente la documentación asociada al procesador especificado en formato HTML.
- ANTLR [124] es un generador de procesadores de lenguajes especificados mediante esquemas de traducción. Los analizadores generados se especifican mediante las anteriormente citadas gramáticas LL(*), que, como ya se ha indicado, son unas gramáticas que permiten variar dinámicamente el número de símbolos necesarios para decidir las producciones a expandir. Además, como JavaCC, permite a los usuarios utilizar la notación EBNF para realizar sus especificaciones, lo que clarifica las especificaciones procesadas por ANTLR. La figura 23 muestra una codificación ANTLR de un lenguaje para la definición de constantes numéricas. La herramienta es capaz de generar los procesadores en diferentes lenguajes de programación como Java, C, C++, etc. Además, cuenta con herramientas para construir expresamente diferentes estructuras de datos implicadas en el procesamiento (como el árbol de análisis sintáctico) y explorarlas,

proporcionando al usuario información adicional para depurar los procesadores especificados.

2.4.2.2.2 Herramientas de Generación de Traductores Ascendentes

```
...
%union {
int num;
   char *s;
struct symtab *sp;
}

%token <s> ID
%token <num> NUM

%type <sp>decs
%type <sp>dec
%%

decs :  dec ',' decs {$$ = addReg($1, $2);}

decs :  dec {$$ = $1;}

dec :  ID NUM {$$ = crearReg($1, $2);}

%%
```

Figura 24. Fragmento de especificación YACC de un lenguaje de definición de constantes numéricas.

```
action code {:
    private Map<String, Integer>const = new HashMap<String, Integer>();
:}

terminal String ID, NUM;

non terminal decs, dec;

decs ::= decs "," dec {: :};

decs ::= dec {: :};

dec ::= id:ID num:NUM {: const.put(id, Integer.parseInt(num));:};
```

Figura 25. Especificación CUP de un lenguaje de definición de constantes numéricas.

Las herramientas que generan traductores ascendentes están basadas en esquemas de traducción *bottom-up*. YACC [154], Bison [97], CUP [14], Tatoo [33], SableCC [52], Beaver[2], Copper [173] o YaJco[3], son algunos ejemplos de este tipo de

[2] http://beaver.sourceforge.net/

generadores de traductores. A continuación, se describen las características de dos de estas herramientas: YACC y CUP.

- YACC (*Yet Another Compiler-Compiler*) [98] es un generador de traductores ascendentes en código C. Las especificaciones que es capaz de analizar están basadas en gramáticas LALR, en formato BNF, en las que los desarrolladores tendrán que insertar las acciones semánticas. Esta herramienta fue desarrollada en 1970, y aún hoy en día es utilizada. La figura 24 muestra un fragmento de una especificación para el lenguaje de declaración de constantes numéricas en YACC. La herramienta ha sido ampliada para generar código para otros lenguajes como Java, Python o Ruby. YACC no es capaz de generar el analizador léxico por su cuenta, y se vale de generadores de analizadores léxicos externos e independientes como Lex [98]o Flex [126].
- CUP (*Construction of Useful Parsers*) [14] [72] es una herramienta similar a YACC, que genera implementaciones Java de los traductores especificados. La figura 25 muestra una codificación CUP del lenguaje para la especificación de constantes numéricas. Al igual que YACC, CUP opera con especificaciones basadas en gramáticas LALR, y tampoco soporta la generación del analizador léxico. Así mismo, en cierto sentido es más limitado que YACC (v.g., las acciones semánticas en YACC permiten referir a valores semánticos en posiciones arbitrarias de la pila, lo que, en algunas ocasiones, permite emular herencia de atributos; por su parte, CUP no ofrece notación específica al respecto). No obstante, el metalenguaje de CUP es, en cierta medida, más claro y legible que el soportado por YACC, lo que realza el valor educativo de CUP frente al de YACC.

2.4.3 Implementación de especificaciones basadas en gramáticas de atributos mediante esquemas de traducción

En esta tesis se propone utilizar herramientas convencionales de generación de traductores (basadas en esquemas de traducción) para llevar a cabo la implementación de especificaciones basadas en gramáticas de atributos, con el fin de permitir la mejora de los procesos de creación de las especificaciones mediante el prototipado rápido e implementación sistemática de las mismas. A este respecto, existen diferentes técnicas que permiten utilizar las herramientas de generación de traductores basadas en esquemas de traducción para implementar procesadores de lenguaje a partir de especificaciones basadas en gramáticas de atributos. Estas técnicas normalmente se basan en el acoplamiento de la evaluación de atributos semánticos con el proceso de análisis sintáctico, acoplamiento que ha sido extensivamente abordado como una forma de implementar diferentes clases de gramáticas de atributos (véase [7] para un tutorial introductorio). Las técnicas en sí pueden clasificarse en dos grupos, dependiendo del tipo de generador utilizado: técnicas orientadas a traductores descendentes, y técnicas orientadas a traductores ascendentes. A continuación se revisan cada una de ellas.

[3] http://code.google.com/p/yajco/

2.4.3.1 Técnicas orientadas a traductores descendentes

Los trabajos [5] [7] muestran cómo utilizar generadores de traductores descendentes para implementar gramáticas LL-*atribuidas*: gramáticas de atributos l-atribuidas formuladas sobre gramáticas incontextuales LL. La técnica consiste en anteponer el cálculo de los atributos heredados de cada símbolo en la parte derecha de una producción inmediatamente antes de dicho símbolo, así como situar el cálculo de los atributos sintetizados de la parte izquierda al final de la producción. La implementación de gramáticas más generales supone transformar la especificación original, aplicando mecanismos y transformaciones por las cuales se puede transformar una gramática de atributos en una gramática LL-atribuida equivalente. Una vez transformada la gramática, el desarrollador es capaz de utilizar la especificación transformada como entrada de las herramientas de generación de traductores descendentes para obtener el procesador especificado.

Algunas de las transformaciones orientadas a la obtención de gramáticas LL-atribuidas equivalentes a una especificación inicial son:

- La eliminación de la recursión a izquierdas y factorización [5]. Estas transformaciones únicamente pueden sistematizarse en el caso de gramáticas s-atribuidas (es decir, que involucren únicamente atributos sintetizados). La presencia de atributos heredados debe, por tanto, manejarse de manera específica para cada caso, lo que introduce un factor de error significativo en el proceso.
- Transformaciones de *parcheo* [5]. Estas transformaciones están orientadas a manejar especificaciones no l-atribuidas (es decir, con dependencias por la derecha). La idea básica es dejar *huecos* en lugar de los valores provenientes de dependencias por la derecha, y substituir dichos huecos con los valores reales conforme estos se van determinando. Estas transformaciones son específicas para cada especificación, lo que dificulta también su sistematización, con el consiguiente riesgo de introducción de errores durante el proceso. No obstante, el uso de lenguajes de programación lógica, como Prolog [160], para implementar las especificaciones puede permitir manejar ciertas dependencias por la derecha (pero no cualquiera, ya que la técnica se restringe a las denominadas *logical one-pass attribute grammars* [122] [121]).

2.4.3.2 Técnicas orientadas a traductores ascendentes

Al igual que ocurre con las herramientas de generación de traductores descendentes, es posible construir procesadores de lenguaje a partir de gramáticas de atributos utilizando herramientas de generación de traductores ascendentes. Aunque en los modelos más generales de acoplamiento de la evaluación semántica en gramáticas de atributos con el análisis sintáctico (véase, por ejemplo [7] [8] [107]) es posible ejecutar operaciones de evaluación tanto durante las acciones de reducción como durante las acciones de desplazamiento (lo que permite implementar la clase de gramáticas denominadas LR atribuidas), la mayoría de las herramientas de generación de traductores ascendentes permiten únicamente asociar acciones semánticas con la reducción de las reglas. Esto permite únicamente la implementación directa de gramáticas de atributos s-atribuidas

formuladas sobre gramáticas incontextuales LALR, en lugar de gramáticas LR atribuidas más generales. Efectivamente, los valores de los atributos sintetizados de las partes izquierdas se calcularán en las citadas acciones asociadas a la reducción de las reglas. No obstante, en ciertas situaciones es posible implementar también gramáticas con atributos heredados, mediante la técnica de *marcadores* [5]. Esta técnica se basa en introducir nuevos no terminales (*marcadores*) definidos mediante una producción con parte derecha vacía, y utilizar los mismos para almacenar atributos heredados. Para ello, la gramática incontextual debe modificarse situando los marcadores en los lugares adecuados de las producciones, lo que, a su vez, puede redundar en la pérdida del carácter LALR de la gramática. Este hecho convierte la técnica en una técnica difícil de aplicar y de sistematizar.

2.4.4 Usos educativos de las herramientas para el desarrollo de procesadores de lenguaje

Las secciones anteriores evidencian cómo las herramientas profesionales para el desarrollo de procesadores de lenguaje son de gran utilidad en el desarrollo práctico de traductores. No obstante, en el ámbito educativo estas herramientas no tienen necesariamente que proporcionar las características adecuadas para las necesidades de los estudiantes. Aun así, existen diferentes estrategias educativas que utilizan estas herramientas para presentar diferentes conceptos y técnicas a los estudiantes. De hecho, en el apartado 2.2 ya se ha mostrado cómo las herramientas de construcción de procesadores de lenguaje pueden jugar un papel importante en el proceso de enseñanza-aprendizaje de la materia. No obstante, mientras que en los enfoques discutidos en dicho apartado las herramientas se utilizaron como instrumentos auxiliares para generar procesadores susceptibles de ser aumentados mediante técnicas de simulación, estas herramientas pueden jugar también un papel central en el proceso de enseñanza-aprendizaje, tal y como se discute en las siguientes subsecciones.

2.4.4.1 Usos educativos de las herramientas genéricas

Como ya se ha indicado, las herramientas de desarrollo genéricas de procesadores de lenguaje se utilizan ampliamente para el desarrollo profesional de procesadores de lenguaje. Además, dentro de unos parámetros y condiciones determinadas, como ya se ha indicado también, estas herramientas se pueden usar con fines educativos dentro de las materias de Procesadores de Lenguajes. A continuación se describen dos trabajos donde se emplean estas herramientas con un carácter educativo.

2.4.4.1.1 LISA

LISA, herramienta basada en gramáticas de atributos ya descrita en la subsección 2.4.1, ha sido utilizada también con propósito educativo. Este uso viene motivado por las diferentes visualizaciones y simuladores proporcionados por la herramienta, que cuenta con un depurador del proceso de evaluación semántica, un simulador de la construcción del árbol sintáctico, un visualizador del grafo de

dependencias, etc. A continuación, se describen estas funciones para evidenciar su potencial educativo:

- El visualizador de los grafos de dependencias muestra, por cada regla sintáctica, cómo se relacionan los atributos semánticos de cada categoría sintáctica implicada en la producción.
- El simulador de la construcción del árbol sintáctico muestra de un solo vistazo el árbol completo, y mediante una animación muy simple, cómo se construye cada nodo del árbol y cómo se van relacionando para llegar al árbol resultante.
- El depurador del proceso de evaluación semántica muestra, sobre los nodos del árbol sintáctico, el cambio de valor de las instancias de atributo durante el proceso de evaluación. Para ello, los nodos son representados como tablas de pares atributo/valor donde cada fila representa una instancia de atributo asociada a la categoría sintáctica a la que pertenece dicho nodo. El depurador, a medida que se realiza el proceso de evaluación, marca el nodo y la instancia de atributo implicado, así como la ecuación semántica aplicada en la gramática de atributos, y seguidamente muestra el cambio de valor sufrido por la aplicación de dicha ecuación. La figura 26 muestra una captura de este simulador en funcionamiento.

En [109] se muestran los resultados de diferentes estudios realizados entre profesores y estudiantes. Los alumnos valoraron positivamente el entorno, ya que las características de depuración visual de la herramienta les resulto útil a la hora de analizar el funcionamiento de los diferentes algoritmos de análisis disponibles para las fases léxica, sintáctica y semántica. Por su parte, los docentes destacaron la motivación percibida en los alumnos al usar la herramienta.

2.4.4.1.2 ANTLRWorks

ANTLRWorks [26] es un entorno de desarrollo para ANTLR. ANTLRWorks proporciona un editor de gramáticas, un intérprete para ayudar al usuario durante la fase de prototipado de la gramática y un depurador para identificar errores en la especificación. El depurador proporcionado por ANTLRWorks es muy completo permitiendo mostrar la construcción, de forma dinámica, del árbol sintáctico y del árbol sintáctico decorado. Además, permite determinar *breakpoints* a nivel gramatical, en la sentencia o por eventos del procesador. Por otro lado, el depurador es capaz de identificar problemas de ambigüedad en las gramáticas, y ayudar al usuario a solucionarlos. La figura 27 presenta una captura del sistema, mostrando la visualización del árbol sintáctico. De esta forma, los alumnos son capaces de depurar los procesadores exigidos en clase.

2.4.4.2 Herramientas específicas para el ámbito educativo

Estas herramientas son herramientas para la generación de procesadores de lenguaje, pero esta vez explícitamente dedicadas a su uso en educación. Como ejemplos de este tipo de entornos, se describen VCOCO y PAG.

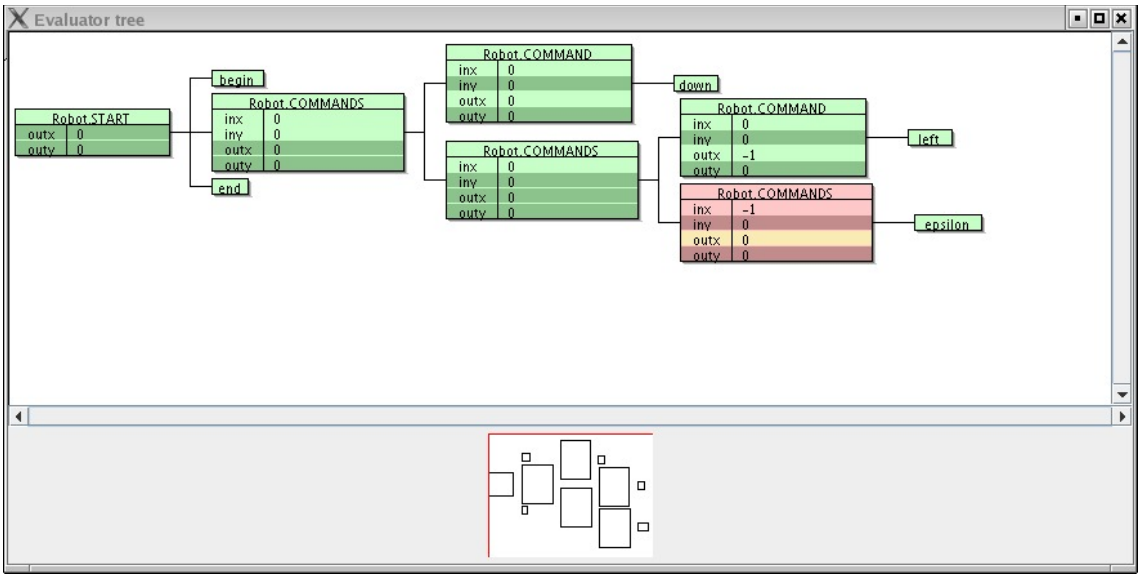


Figura 26. Captura del simulador de la evaluación semántica proporcionado por LISA.

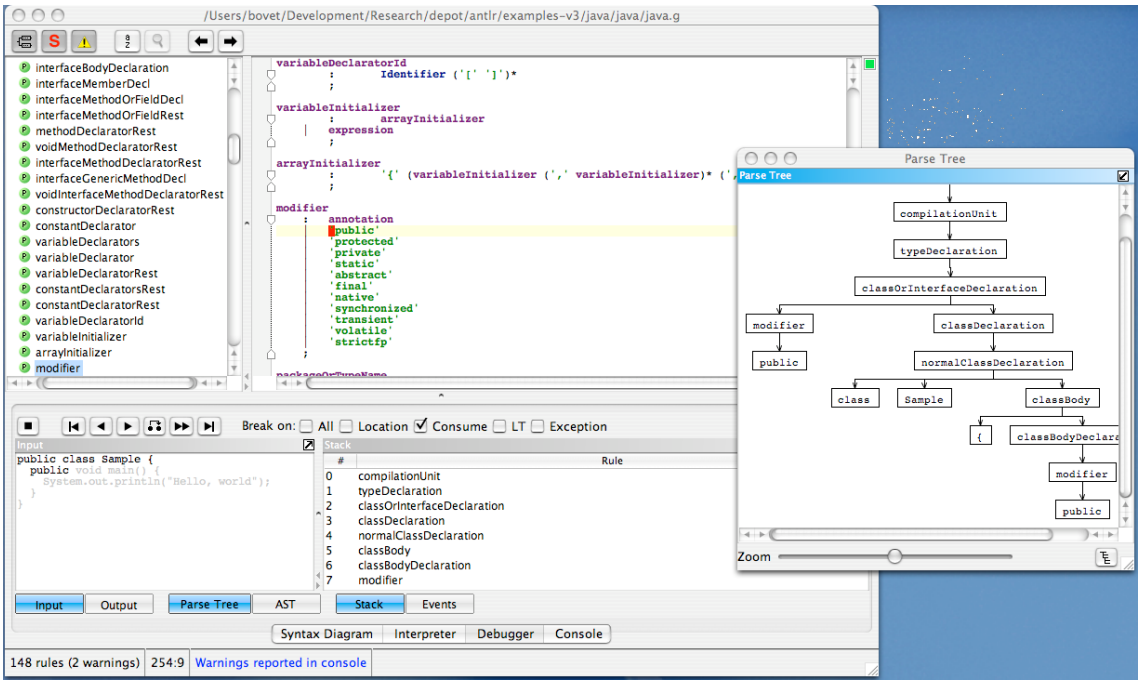


Figura 27. Captura de la herramienta ANTLRWorks

2.4.4.2.1 VCOCO

VCOCO [130] es un software de visualización para el lenguaje de creación de procesadores COCO, un lenguaje similar a JavaCC o YACC. COCO genera el procesador a partir de la especificación de lenguajes mediante gramáticas EBNF anotadas con acciones semánticas. El simulador muestra al alumno la gramática, la sentencia, el código maquina generado y el código fuente del programa que se está depurando. La herramienta, durante la simulación de un procesador, resalta las diferentes partes implicadas en cada paso de simulación para que los alumnos sean

capaces de comprender el funcionamiento de todas las partes del procesador como unidad. Este funcionamiento se puede observar en la figura 28.

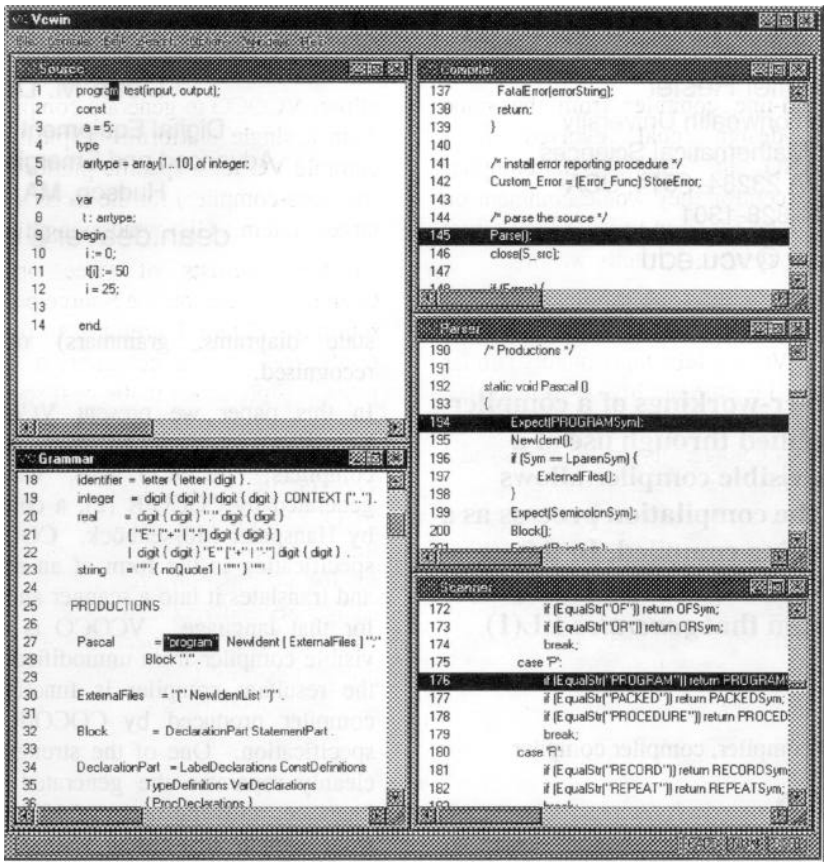


Figura 28. Captura de VCOCO (extraída de [130])

2.4.4.2.2 PAG

PAG (*Prototyping with Attribute Grammars*) [158] es un entorno de desarrollo que es capaz de procesar especificaciones basadas en gramáticas de atributos para crear prototipos de procesadores de lenguaje. Las especificaciones se escriben empleando una sintaxis embebida en lenguaje Prolog, de una forma muy similar a la de las gramáticas de atributos. Aparte de la gramática de atributos (o fragmentos de estas), el usuario debe especificar las funciones semánticas necesarias. La herramienta genera entonces el prototipo de procesador, que al analizar una frase permitirá inspeccionar todos los posibles árboles sintácticos decorados (PAG funciona con gramáticas incontextuales arbitrarias, incluso gramáticas ambiguas). PAG permite al usuario navegar por los nodos de cada árbol sintáctico generado, donde se mostrarán, de forma estática, las instancias de atributos asociadas al nodo seleccionado. La información mostrada para cada atributo será su nombre, su valor semántico y la expresión semántica utilizada para calcular dicho valor. La figura 29 muestra una captura de un prototipo de procesador en funcionamiento.

Esta herramienta ha sido empleada con fines educativos en dos ámbitos distintos: en la asignatura de Procesadores de Lenguaje dentro de Informática, y en la materia de Lingüística Computacional en Filología.

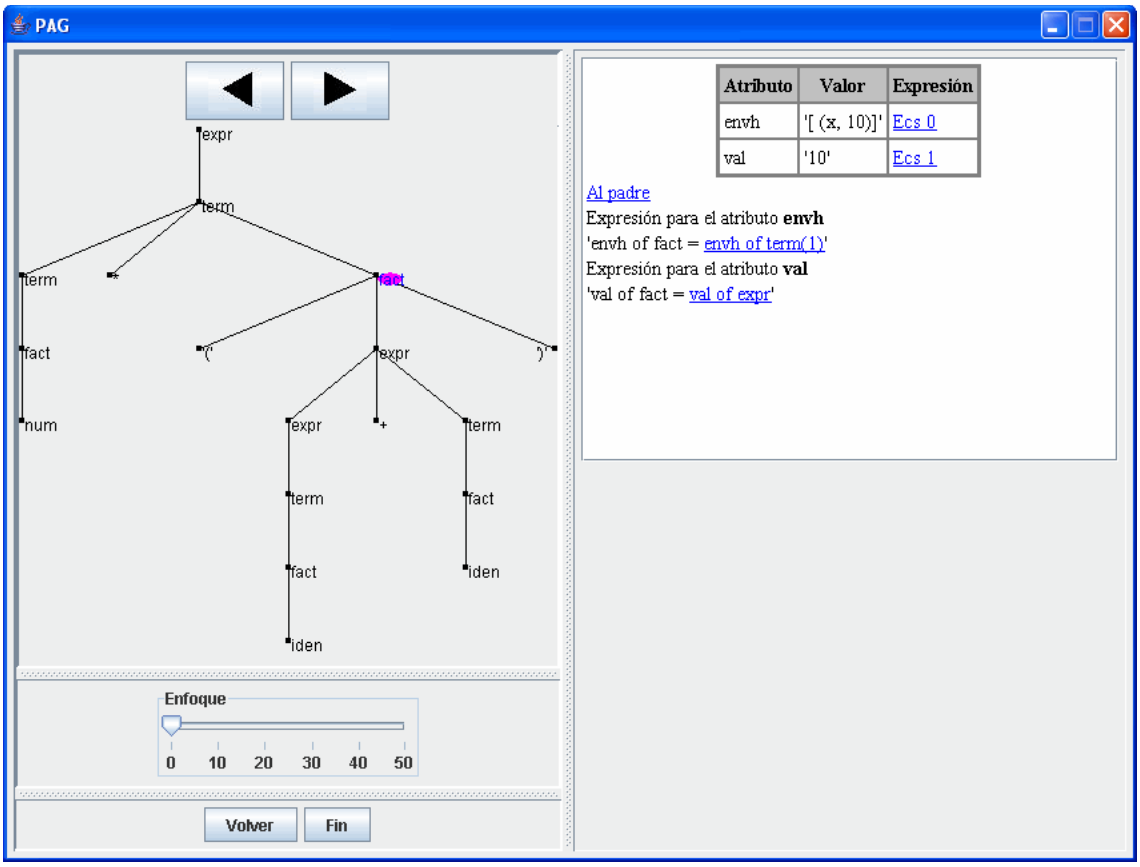


Figura 29. Captura de la herramienta PAG (tomada de [158])

2.4.5 Discusión

En este apartado se ha podido constatar la existencia de un amplio abanico de herramientas para la generación de procesadores basadas en gramáticas de atributos y en esquemas de traducción. A este respecto:

- Las herramientas de generación de procesadores basadas en gramáticas de atributos proporcionan un marco adecuado para el desarrollo de procesadores de lenguaje dirigido por especificaciones declarativas. Sin embargo, la implementación generada no se corresponde fielmente a la especificación de partida, debido a diferentes optimizaciones que se realizan durante la generación.
- Por otro lado, las herramientas para la generación de procesadores basadas en esquemas de traducción soportan formalismos de más bajo nivel, más orientados a describir la implementación de los procesadores mediante formalismos específicos del dominio (ligados a las gramáticas) que a especificar declarativamente dichos procesadores.

Además, se han analizado diferentes técnicas y estrategias de implementación de procesadores de lenguajes basados en gramáticas de atributos empleando el potencial de los generadores basados en esquemas de traducción. Estas técnicas y estrategias de implementación tienen un interés indudable desde un punto de vista educativo, ya que permiten al alumno entender cómo convertir sistemáticamente la especificación de un procesador de lenguaje en su implementación final. Sin embargo, los enfoques analizados presentan como principal restricción el ser

capaces de manejar únicamente tipos particulares de gramáticas de atributos (gramáticas LL-atribuidas en el caso de generadores de traductores descendentes, gramáticas s-atribuidas definidas sobre gramáticas incontextuales LALR en el caso de traductores ascendentes). Si bien es posible tratar con gramáticas más generales, normalmente será necesario aplicar técnicas específicas para cada especificación particular, práctica normalmente propensa a errores y que, por tanto, puede dificultar el aprendizaje final de los estudiantes.

Se ha evidenciado también el valor educativo de las herramientas de generación de procesadores en cursos típicos de Procesadores de Lenguaje con resultados positivos para los estudiantes. Estos resultados se han obtenido gracias a diferentes generadores de procesadores que proporcionan a los desarrolladores diferentes depuradores enriquecidos y visualizaciones que permiten entender el procesamiento de una forma más intuitiva. El uso educativo de LISA y ANTLRWorks constituyen dos ejemplos de esta estrategia. Por otro lado, existen generadores de procesadores orientados a la educación, cómo PAG y VCOCO, que están específicamente diseñados para ayudar a los estudiantes a entender el funcionamiento de sus propios procesadores de lenguaje mediante el uso de simuladores. Sin embargo, desde nuestro punto de vista, estas herramientas presentan las siguientes deficiencias:

- Las herramientas basadas en esquemas de traducción, tanto ANTLRWorks como VCOCO, se centran especialmente en aspectos relativos a la implementación de los procesadores, dejando de lado aspectos relativos a la especificación de los mismos, utilizando formalismos de más alto nivel.
- Las herramientas basadas en el formalismo de las gramáticas de atributos (v.g., LISA, PAG) no permiten involucrar interactivamente al alumno en el proceso de evaluación semántica, lo que, desde nuestro punto de vista, es un aspecto esencial para potenciar la comprensión de las especificaciones. Además, tampoco ofrecen mecanismos que permitan al alumno entender cómo convertir las especificaciones en implementaciones efectivas de los procesadores, utilizando, para ello, estrategias de implementación sistemáticas. Por el contrario, los traductores generados se basan en implementaciones en las que la conexión entre la especificación y la implementación se pierde (v.g., implementaciones altamente optimizadas de los procesos de evaluación semántica [69] en el caso de las herramientas basadas en gramáticas de atributos como LISA, un intérprete genérico de gramáticas de atributos escrito en Prolog en el caso de PAG [158]).

2.5 A modo de conclusión

A lo largo de este capítulo se han analizado diferentes aspectos de la enseñanza y el aprendizaje de conceptos relacionados con el diseño y la implementación de lenguajes informáticos: estrategias de enseñanza-aprendizaje de las materias de Procesadores de Lenguaje, uso de simulaciones educativas y juegos serios, y herramientas específicas para la construcción de los procesadores.

Las estrategias de enseñanza de procesadores de lenguajes analizadas siguen diferentes enfoques para transmitir a los alumnos los conceptos y técnicas necesarias para el diseño y desarrollo de un procesador de lenguaje. La estrategia

más ampliamente utilizada toma como piedra angular el desarrollo, por parte de los alumnos, de un procesador para un lenguaje de programación. Este objetivo es el hilo conductor de las clases teóricas y las sesiones de laboratorio. Por su parte, otras estrategias tratan de motivar a los alumnos proponiendo proyectos atractivos basados en otro tipo de lenguajes, o presentan los conceptos de una forma radicalmente distinta a la tradicional, mediante la experimentación con procesadores de lenguaje reales. Si bien es cierto que los diferentes enfoques analizados tienen sus ventajas y desventajas, dentro del marco de esta tesis todas las estrategias educativas tienen una carencia común muy importante para la enseñanza de la construcción de procesadores de lenguaje: dichas estrategias se centran principalmente en la parte de desarrollo de los procesadores, en detrimento de la parte previa de especificación, fase esencial para articular un correcto desarrollo.

En lo referente a las simulaciones interactivas, en este capítulo se ha analizado cómo diferentes herramientas software aprovechan las ventajas educativas que proporcionan las visualizaciones y la interacción para mejorar la experiencia educativa de los estudiantes, ayudándoles a entender los diferentes procesos y artefactos que forman parte del dominio educativo. En relación con el uso de simulaciones educativas en el campo de los Procesadores de Lenguaje se ha evidenciado, sin embargo, como la mayoría de las herramientas analizadas se centran en aspectos relativos a la implementación de dichos procesadores, observándose una importante carencia de simuladores centrados en los aspectos de la especificación.

En lo referente a los juegos serios, el capítulo ha evidenciado cómo estos enfoques aprovechan el gran atractivo de este tipo de artefactos para motivar a los estudiantes a aprender diferentes conceptos de una forma amena. Aunque el abanico de este tipo de software en la enseñanza de la informática es amplio, no ocurre lo mismo para la construcción de procesadores de lenguaje. Esto es debido, principalmente a la dificultad de encontrar una mecánica de juego adecuada para los conceptos que componen el curso que sea, a su vez, entretenida y atractiva para los alumnos. Como excepción se ha analizado el sistema J^2VM que, sin embargo, hace énfasis de nuevo en los aspectos de implementación, en lugar de en las fases previas de la especificación.

Por último, se han analizado diferentes herramientas de generación de procesadores de lenguaje a partir de especificaciones basadas en gramáticas de atributos y esquemas de traducción. Estas herramientas son adecuadas para desarrolladores experimentados en diseño e implementación de lenguajes, ya que les permiten desarrollar de forma rápida y efectiva complejos procesadores de lenguaje. En este capítulo también se ha hecho patente el potencial educativo de dichas herramientas, sobre todo de aquellas que proporcionan a los alumnos visualizaciones interactivas de diferentes estructuras de datos implicadas en el procesamiento del lenguaje (árbol de análisis sintáctico, pila de análisis, etc.). No obstante, las herramientas basadas en esquemas de traducción están más orientadas a la implementación de los procesadores que a su especificación. Por su parte, aunque las herramientas basadas en gramáticas de atributos operan sobre especificaciones de más alto nivel, se ha observado que, por una parte, no permiten involucrar interactivamente al alumno en el proceso de evaluación semántica, y

que, por otra, no generan implementaciones que preserven la especificación. Este segundo aspecto, la generación de implementaciones que preserven la especificación, puede ser abordado mediante técnicas de implementación de gramáticas de atributos sobre esquemas de traducción, aunque, tal y como se ha discutido también en este capítulo, la aplicación sistemática de dichas técnicas, en su forma actual, está restringida a tipos particulares de gramáticas de atributos, mientras que el manejo de otros tipos más generales precisa de transformaciones no triviales y propensas a errores.

A modo de conclusión, los diferentes enfoques, estrategias y herramientas software analizadas en el ámbito de la construcción de procesadores de lenguaje ponen de manifiesto que, desde un punto de vista educativo, existe una importante carencia en la atención que se presta a la enseñanza de los conceptos relacionados con la especificación de procesadores de lenguaje, frente a las técnicas de implementación de los mismos. Esta tesis pretende dar solución parcial a dicha carencia mediante: (i) la formulación y soporte informático de una estrategia que facilite la comprensión de los mecanismos básicos subyacentes a las especificaciones basadas en gramáticas de atributos en etapas tempranas del proceso de enseñanza-aprendizaje, (ii) la implementación sistemática de estas especificaciones utilizando herramientas basadas en esquemas de traducción, y (iii) la integración de los dos aspectos anteriores en una herramienta orientada al desarrollo educativo de procesadores de lenguaje dirigida por especificaciones y basada en la depuración.

Capítulo 3: Objetivos y planteamiento del trabajo

Tal y como ya se ha indicado en los capítulos anteriores, el correcto desarrollo de un procesador de lenguaje depende fuertemente de: (i) una fase inicial de especificación correctamente ejecutada, en la que los diferentes aspectos del procesador se describan rigurosamente y de forma no ambigua utilizando formalismos específicamente dirigidos a cada uno de estos aspectos, y (ii) la aplicación sistemática de técnicas de implementación dirigidas por las especificaciones, que permitan transformar tales descripciones formales en la implementación final del procesador. No obstante, muchas veces los alumnos que se inician en las materias de diseño e implementación de lenguajes informáticos carecen de la destreza necesaria para producir especificaciones adecuadas, y, por tanto, abordan el desarrollo de sus procesadores basándose en diseños incompletos o incorrectos. Este hecho es especialmente crítico en lo que se refiere a aspectos clave en el desarrollo de un procesador, como la especificación de procesos de traducción dirigidos por la sintaxis utilizando, por ejemplo, un formalismo basado en gramáticas de atributos. Por otro lado, llegada la fase de implementación, los alumnos muchas veces obvian las especificaciones, abordando el desarrollo de forma arbitraria, apoyándose en su experiencia como programadores, en lugar de en la aplicación sistemática de técnicas de implementación dirigidas por dichas especificaciones.

De esta forma, es necesario que los alumnos de las materias relativas al desarrollo de procesadores de lenguaje asimilen correctamente la importancia de las especificaciones en dicho desarrollo, y sean capaces de entenderlas, producirlas y aplicarlas de forma efectiva durante la construcción de los procesadores. Sin embargo, tal y como se ha evidenciado en el capítulo anterior, existe un vacío claro al respecto en relación con las diferentes estrategias de enseñanza y desarrollo de procesadores de lenguaje, así como con las diferentes herramientas software para la enseñanza de esta materia (v.g., simuladores y juegos serios), ya que estas se

centran principalmente en aspectos de implementación y funcionamiento interno de algoritmos y procesadores, en detrimento de los aspectos de especificación anteriormente citados. Por ello, esta tesis propone abordar dicho vacío relativo a herramientas y estrategias que incidan en la comprensión, producción y explotación de las especificaciones durante el proceso de enseñanza-aprendizaje de las citadas materias sobre procesadores de lenguaje.

De esta forma, en este capítulo se presentan los objetivos que se pretenden alcanzar en la tesis planteada (apartado 3.1). Seguidamente, se describe el plan de trabajo que se seguirá para lograr dichos objetivos (apartado 3.2).

3.1 Objetivos de la tesis

Tal y como se ha mencionado anteriormente, frecuentemente los alumnos de los cursos introductorios a los procesadores de lenguaje no consideran importantes los aspectos relativos a la especificación y al desarrollo sistemático basado en la especificación. Este hecho se evidencia, por ejemplo, en el desarrollo de los procesadores de lenguaje planteados como proyectos finales de dichos cursos, donde es frecuente observar cómo la tarea de especificar y desarrollar dichos procesadores resulta tediosa e insatisfactoria para los alumnos, debido a la gran cantidad de problemas que encuentran durante el desarrollo, al basar este en métodos *ad hoc* en lugar de en métodos sistemáticos focalizados en las especificaciones. Teniendo en cuenta estas consideraciones, el trabajo planteado en esta tesis se centra en proporcionar estrategias educativas y herramientas software para mejorar el entendimiento de los formalismos de especificación de procesadores de lenguaje, y asistir a los alumnos en la aplicación de las técnicas de desarrollo sistemático de procesadores de lenguaje a partir de su especificación. En particular, y dada la importancia que cobra la especificación de los procesos de traducción dirigida por la sintaxis en el desarrollo de los procesadores, esta tesis centrará su atención en el formalismo de las gramáticas de atributos. Para ello, esta tesis plantea los siguientes tres objetivos:

- El primer objetivo se centra en desarrollar una estrategia educativa para facilitar a los estudiantes la comprensión de los conceptos básicos en la especificación de un procesador de lenguaje mediante el formalismo de las gramáticas de atributos. Dicha estrategia estará focalizada en etapas tempranas del aprendizaje, y será soportada por un conjunto de herramientas software que se apoyarán en las nuevas tendencias en eLearning para presentar dichos conceptos (en concreto en juegos serios y simulaciones interactivas). Además, la consecución de este objetivo precisará de diferentes evaluaciones relativas a la adecuación pedagógica de la estrategia de enseñanza-aprendizaje y de las herramientas software asociadas.
- El segundo objetivo se centra en proporcionar una estrategia de desarrollo de procesadores de lenguaje que, partiendo de una especificación en forma de gramática de atributos no circular general, permita obtener prototipos funcionales del procesador de lenguaje especificado. Dichos prototipos estarán soportados por herramientas convencionales para la generación de

procesadores, basadas en esquemas de traducción. Adicionalmente, se evaluará la utilidad pedagógica de este enfoque.

- Por último, el tercer objetivo se centra en proporcionar una plataforma software para el desarrollo de procesadores de lenguaje a partir de su especificación mediante el formalismo de las gramáticas de atributos que integre los principales resultados obtenidos en los dos objetivos anteriores: por una parte, la estrategia de comprensión del formalismo de las gramáticas de atributos, y, por otra parte, la codificación de gramáticas de atributos sobre herramientas convencionales de desarrollo de procesadores de lenguaje. Utilizado la estrategia de comprensión, dicha plataforma ofrecerá herramientas avanzadas para la depuración de los procesadores de lenguaje especificados mediante gramáticas de atributos. Por otro lado, la herramienta utilizará los patrones de codificación de gramáticas de atributos no circulares generales sobre esquemas de traducción para generar dichos procesadores. Además, este tercer objetivo también se centrará en evaluar la utilidad educativa de la plataforma desarrollada.

Las siguientes secciones profundizan en cada uno de estos aspectos, sentando las bases para el planteamiento del trabajo de tesis, que se esboza en el siguiente apartado.

3.1.1 Estrategia para la comprensión de las especificaciones basadas en gramáticas de atributos

Los formalismos de especificación de procesadores de lenguajes son, tradicionalmente, una de las piedras angulares a utilizar durante un curso de procesadores de lenguaje. Los alumnos necesitan manejarlos con soltura, pues se utilizarán durante todo el curso para especificar y diseñar los diferentes componentes que componen dicho procesador (analizadores léxicos y sintácticos, construcción de la tabla de símbolos, comprobación de las restricciones contextuales, generación de código, etc.). Este aspecto de la tesis se centra específicamente en el formalismo de las gramáticas de atributos analizado en el capítulo anterior. Normalmente, este formalismo permite especificar la sintaxis de un lenguaje, así como su semántica. Además, presenta un proceso de cómputo característico, dirigido por las dependencias entre atributos, mediante el cual se calculan los valores semánticos asociados con las estructuras sintácticas. Este aspecto de las gramáticas de atributos suele resultar especialmente difícil de asimilar para los alumnos.

Tal y como hemos visto en el capítulo anterior, existen diferentes estrategias y herramientas software destinadas a mejorar el proceso de enseñanza-aprendizaje de la materia de Procesadores de Lenguaje. Sin embargo:

- Aunque las diferentes estrategias analizadas en el capítulo anterior han demostrado su valor educativo en el ámbito de la enseñanza de dicha materia, dichas estrategias adolecen de un aspecto esencial desde el punto de vista del enfoque principal de esta tesis: el foco de las mismas se centra en aspectos relacionados con la implementación de los procesadores, dejando en un segundo plano la parte de especificación de dichos

procesadores, parte en la que, sin embargo, se apoya todo el proceso de desarrollo ulterior.

- Lo mismo ocurre con las herramientas software analizadas, tanto los simuladores interactivos como los juegos serios relacionados con la enseñanza de Procesadores de Lenguaje. Ambos están fuertemente orientados a aspectos directamente relacionados con la implementación de los procesadores de lenguajes.

Por ello, teniendo en cuenta la importancia de los conceptos básicos de especificación basados en el formalismo de las gramáticas de atributos desde las primeras fases del proceso de enseñanza-aprendizaje y las dificultades que pueden encontrar los alumnos tanto en su aprendizaje, como en su aplicación en el diseño y desarrollo de un procesador de lenguaje, el primer objetivo de esta tesis se formula como:

> *Objetivo 1. Formular una estrategia educativa para facilitar la comprensión del formalismo de las gramáticas de atributos en fases tempranas del aprendizaje, soportar dicha estrategia mediante herramientas software educativas basadas en juegos serios y simulaciones interactivas, y validar la adecuación pedagógica tanto de la estrategia como de las herramientas software mediante evaluaciones con docentes y estudiantes de las materias de Procesadores de Lenguaje.*

3.1.2 Estrategia para la implementación de gramáticas de atributos como esquemas de traducción

Otro de los aspectos más problemáticos a los que se enfrentan los alumnos de un curso de Procesadores de Lenguaje es obtener la implementación de un procesador a partir de su especificación. Este proceso puede entrañar transformaciones no triviales, por lo que los alumnos pueden cometer múltiples errores que redundan en una implementación final incorrecta, desligada de la especificación original.

En concreto, y tal y como se ha indicado en el capítulo anterior, en el caso de especificaciones basadas en gramáticas de atributos, aunque existen herramientas de generación de procesadores de lenguajes basadas directamente en dicho formalismo, dichas herramientas realizan de forma automática el proceso de generación a partir de las gramáticas de atributos de entrada, resultando de esta forma dicho proceso completamente opaco para los estudiantes, y no existiendo una relación directa fácilmente identificable entre las implementaciones generadas y las especificaciones proporcionadas.

Por tanto, desde un punto de vista didáctico es importante que los estudiantes aprendan cómo relacionar sus especificaciones basadas en gramáticas de atributos con los modelos de traducción más significativos. Para ello, y tal y como se ha analizado en el capítulo anterior, el enfoque más habitual es aplicar secuencialmente transformaciones a las gramáticas de atributos originales con el fin de obtener esquemas de traducción equivalentes que se pueden implementar directamente utilizando herramientas de generación de procesadores más convencionales, como CUP o JavaCC. No obstante, y tal y como se ha mostrado también en el capítulo anterior, dichas transformaciones pueden sistematizarse

únicamente para ciertos tipos de gramáticas de atributos, requiriendo el tratamiento de otros casos de soluciones *ad hoc* y muy dependientes de las especificaciones particulares. El resultado es un proceso muy propenso a errores, que puede desvirtuar el objetivo didáctico final: permitir a los alumnos relacionar especificaciones basadas en gramáticas de atributos con modelos de traducción convencionales. Una vez disponibles las implementaciones de las gramáticas de atributos como esquemas de traducción, los alumnos podrán aplicar transformaciones orientadas a mejorar la eficiencia de las mismas. No obstante, el aspecto crítico en este proceso es encontrar los citados patrones de codificación de gramáticas de atributos no circulares arbitrarias como esquemas de traducción. Por tanto, el segundo objetivo de la tesis se formula de la siguiente forma:

> *Objetivo 2. Formular una estrategia que permita a los alumnos obtener prototipos funcionales de los procesadores de lenguaje mediante la codificación directa de gramáticas de atributos no circulares arbitrarias como esquemas de traducción, dar soporte a dicha estrategia mediante una biblioteca software apropiada, y evaluar la utilidad educativa de la estrategia formulada y de la biblioteca que la soporta.*

3.1.3 Entorno educativo de desarrollo de procesadores de lenguaje basado en gramáticas de atributos

La adecuada asimilación del proceso de desarrollo de procesadores de lenguaje dirigido por especificaciones basadas en gramáticas de atributos requiere, tal y como se ha indicado en las secciones anteriores, de la conjunción de dos aspectos básicos: (i) una profunda comprensión de los conceptos y mecanismos subyacentes a las gramáticas de atributos como formalismo de especificación, y (ii) saber relacionar las especificaciones con sus implementaciones mediante modelos de traducción convencionales. Es, por tanto, importante que los alumnos sean capaces de manejar ambos aspectos con soltura, pero es incluso más importante que los alumnos comprendan cómo se conjugan ambos aspectos para especificar y desarrollar un procesador. Sin embargo, tal y como se ha analizado en el capítulo anterior, las herramientas para la construcción de procesadores de lenguaje no cubren satisfactoriamente dichos aspectos. Efectivamente:

- Las herramientas basadas en esquemas de traducción abordan la descripción de los traductores, en lugar de la descripción de sus especificaciones mediante formalismos de más alto nivel.
- Por su parte, aunque las herramientas basadas en gramáticas de atributos sí aceptan tales especificaciones de más alto nivel, bien no incluyen mecanismos de depuración, bien incluyen mecanismos de depuración orientados a desarrolladores más experimentados. Así mismo, generan implementaciones muy optimizadas que no preservan la especificación.

Por ello, desde un punto de vista didáctico se considera conveniente proporcionar una herramienta para el desarrollo de procesadores de lenguaje a partir de sus especificaciones en forma de gramáticas de atributos que aúne las ventajas de herramientas similares, en lo que se refiere a aceptar directamente las especificaciones, con las ventajas relativas a los aspectos de comprensión e implementación de dichas especificaciones abordados en los objetivos planteados

en las secciones previas. De este modo, el tercer objetivo de la tesis se formula de la siguiente manera:

> *Objetivo 3. Diseñar, implementar y evaluar un entorno de desarrollo de procesadores de lenguaje a partir de sus especificaciones como gramáticas de atributos que integre un modelo de depuración de las especificaciones inspirado en la estrategia de comprensión desarrollada durante la consecución del objetivo 1, y que genere implementaciones de procesadores mediante la aplicación de los patrones de codificación de gramáticas de atributos no circulares arbitrarias como esquemas de traducción desarrollados durante la consecución del objetivo 2*.

3.2 Planteamiento del Trabajo

Teniendo en cuenta las consideraciones anteriores y los objetivos propuestos en la sección anterior, podemos afirmar que el propósito general de esta tesis es proporcionar nuevos enfoques y herramientas para mejorar el proceso de enseñanza/aprendizaje de la especificación de procesadores de lenguaje y su subsiguiente implementación a partir de dicha especificación. Estos enfoques y herramientas aprovecharán los avances descubiertos en pedagogía y *eLearning* para presentar dichos conceptos de una forma atractiva y educativamente efectiva.

De los objetivos planteados en la sección 3.1 se derivan una serie de actividades específicas, que abordan tanto aspectos metodológicos como técnicos, que estructuran el núcleo de este trabajo de tesis, y que cristalizan, finalmente, en la creación de diferentes estrategias educativas y herramientas software que las apoyan. A continuación se describen tales actividades.

3.2.1 Actividades relativas a la estrategia para la comprensión de las especificaciones basadas en gramáticas de atributos

Para lograr la consecución del objetivo 1 es necesario, en primer lugar, formular una estrategia educativa para la enseñanza de los conceptos básicos de la especificación de procesadores de lenguajes mediante el formalismo de las gramáticas de atributos. Después, se desarrollará un sistema software educativo que soporte la estrategia educativa formulada haciendo uso de las nuevas tendencias en eLearning. Posteriormente, se validará tanto la estrategia educativa, como el sistema software educativo, mediante su implantación en el curso. Finalmente, se abstraerá el modelo de proceso seguido en la formulación de la estrategia educativa y del sistema software asociado para permitir dar soporte a otros formalismos de especificación y a otros enfoques diferentes de los basados en las gramáticas de atributos.

3.2.1.1 Formulación de la estrategia

Uno de los principales aspectos a cubrir en el primer objetivo es desarrollar una estrategia educativa para la enseñanza de conceptos básicos del formalismo de las gramáticas de atributos en la especificación de procesadores de lenguajes en las fases iníciales del aprendizaje. Efectivamente, el análisis de las diferentes

estrategias empleadas para la enseñanza de Procesadores de Lenguaje realizado en el capítulo anterior ha evidenciado cómo no existe una estrategia similar específicamente orientada al formalismo de las gramáticas de atributos, con la salvedad de las estrategias implícitas a herramientas como LISA o PAG, herramientas que, por otra parte, presuponen ya conocimientos y destrezas previas en el manejo de dicho formalismo. De esta forma, la estrategia a formular se centrará específicamente en los aspectos relacionados con la comprensión de los conceptos básicos de las gramáticas de atributos, y, en particular, de su proceso de cómputo característico.

El objetivo de la estrategia será conseguir que los alumnos interioricen los mecanismos básicos de evaluación semántica propios del formalismo, y, en particular, el estilo de cómputo derivado de las dependencias entre atributos, y no de la forma concreta de llevar a cabo el análisis sintáctico de las frases del lenguaje. Para ello, dicha estrategia propugnará la inmersión activa del estudiante en el proceso de evaluación semántica, a través de la propuesta de ejercicios específicos relativos a dicha evaluación, en cuya resolución dicho estudiante deberá encontrar posibles maneras de determinar los valores de los atributos en ejemplos clave cuidadosamente seleccionados por el docente.

3.2.1.2 Desarrollo del sistema software de soporte a la estrategia

En este punto, con la estrategia educativa fijada, se planteará el desarrollo de un sistema software educativo que dé soporte a dicha estrategia educativa orientada a mejorar el proceso de enseñanza/aprendizaje del formalismo de las gramáticas de atributos en etapas tempranas de dicha enseñanza/aprendizaje. Así pues, teniendo en cuenta que el software será empleado tanto por docentes y estudiantes, el sistema software deberá proporcionar las herramientas necesarias para apoyar cada uno de los aspectos de la estrategia formulada, y soportar cada uno de los roles implicados en dicha estrategia (docentes y estudiantes). En particular:

- El sistema deberá permitir a los docentes plantear supuestos significativos orientados a mejorar la comprensión de los conceptos básicos subyacentes a las gramáticas de atributos por parte de sus estudiantes.
- El sistema deberá facilitar, así mismo, la conversión de dichos supuestos en artefactos adecuados que ayuden a los estudiantes en el proceso de comprensión de dichos conceptos esenciales. En particular, se buscarán mecanismos que automaticen lo máximo posible la obtención de dichos artefactos.
- El sistema deberá facilitar a los docentes, por último, el analizar el comportamiento de los estudiantes durante el uso de los artefactos aludidos.

Los artefactos en sí permitirán involucrar activamente a los estudiantes en los procesos de evaluación semántica subyacentes a las gramáticas de atributos, en consonancia con la estrategia educativa planteada. Para ello se utilizarán enfoques basados en juegos serios y simulaciones interactivas a fin de propiciar la participación activa de los estudiantes en dichos procesos de evaluación semántica, potenciando, de esta forma, la comprensión de estos aspectos clave del formalismo.

3.2.1.3 Validación de la estrategia y del software asociado

Una vez formulada la estrategia educativa y tras completar el desarrollo del sistema software educativo que dé soporte a dicha estrategia de enseñanza/aprendizaje, es necesario realizar la validación del método educativo para comprobar su utilidad educativa con los estudiantes y docentes de materias relativas al desarrollo de procesadores de lenguaje. Para ello se seguirá el siguiente procedimiento:

- Plantear diferentes evaluaciones con docentes, tanto especialistas en los procesadores de lenguaje, como docentes orientados en otras áreas de la informática relacionada con la Inteligencia Artificial y la Ingeniería del Software. Estas evaluaciones abarcarán diferentes aspectos como: idoneidad de la estrategia educativa, utilidad percibida del sistema software desarrollado, usabilidad de las herramientas software de creación de contenidos y de evaluación de la actividad realizada por los estudiantes, etc.
- Realizar diferentes estudios con estudiantes que cursen materias relativas a los procesadores de lenguaje. En estos estudios se analizará la utilidad percibida tanto de la estrategia educativa, como del sistema software educativo, su eficacia educativa, etc. Plantearemos estos estudios a corto plazo, con experiencias puntuales durante el curso, y a largo plazo, durante varios años académicos para valorar el impacto de la estrategia durante todo el curso.

Teniendo en cuenta que docentes y estudiantes serán los principales implicados tanto en la estrategia educativa, como en el software que lo acompaña, es importante considerar las valoraciones y resultados que ambos aporten durante las experiencias y estudios que se realizarán. Los resultados que se obtengan de dichos estudios proporcionarán información suficiente para poder validar el sistema software desarrollado y la estrategia educativa ideada, así como para poder mejorar dicho sistema y estrategia subyacente.

3.2.1.4 Formulación del modelo de proceso

Con el objetivo de permitir abordar sistemáticamente otros enfoques y formalismos de especificación, se plantea, por último, la abstracción del modelo de proceso básico seguido en el desarrollo del sistema. Dicho modelo de proceso partirá de baterías de ejercicios propuestos por los docentes para reforzar la enseñanza/aprendizaje, y tendrá como objetivo construir, mantener, y hacer evolucionar un sistema educativo que adopte los siguientes sesgos:

- Se aplicará un enfoque generativo que permitirá a los docentes generar el software educativo utilizado por los estudiantes a partir de las baterías de ejercicios.
- Será necesario proporcionar a los docentes herramientas software para editar las baterías de ejercicios que se utilizarán para generar los productos software, así como herramientas para monitorizar y evaluar la actuación de los estudiantes mediante la utilización de los productos software generados.

- Se considerarán diferentes métodos de evaluación de software que permitan a los docentes y desarrolladores guiar el desarrollo del sistema software educativo para que se ajuste a las necesidades de sus usuarios finales.

El modelo en sí surgirá como una abstracción del proceso seguido en el desarrollo del sistema aludido anteriormente, analizándose para ello los distintos aspectos que componen el método de desarrollo seguido, y abstrayendo dichos aspectos para formular el modelo de desarrollo, de forma que este pueda aplicarse a otros contextos relativos a la materia de Procesadores de Lenguaje (e, idealmente, a otras materias en la que tenga sentido plantear estrategias de comprensión de conceptos y procedimientos básicos en etapas tempranas del proceso de enseñanza/aprendizaje).

3.2.2 Actividades relativas a la estrategia para la implementación de gramáticas de atributos como esquemas de traducción

Para abordar el objetivo 2 es necesario, primeramente, formular un conjunto de patrones que permitan codificar gramáticas de atributos no circulares arbitrarias sobre herramientas de desarrollo de procesadores convencionales, basadas en esquemas de traducción. Una vez fijada la estrategia de codificación, se desarrollará una biblioteca que dé soporte software a dicha estrategia y facilite la codificación de las gramáticas sobre las herramientas convencionales de desarrollo. Finalmente, se evaluarán la estrategia formulada y el software desarrollado mediante su aplicación en un curso de Procesadores de Lenguaje, con el fin de validar la adecuación de pedagógica de ambos.

3.2.2.1 Formulación de la estrategia de desarrollo

La principal tarea del objetivo 2 es idear una estrategia de desarrollo para la codificación directa de las especificaciones de procesadores de lenguaje mediante el formalismo de las gramáticas de atributos en herramientas convencionales de construcción de este tipo de procesadores. Mediante la aplicación de dicha estrategia los alumnos serán capaces de obtener prototipos funcionales basados en esquemas de traducción. Efectivamente, en el capítulo anterior se han analizado diferentes enfoques para la implementación de gramáticas de atributos como esquemas de traducción. No obstante, tal y como se ha comentado anteriormente, dichos enfoques adolecen de diversos problemas que restan a los mismos utilidad pedagógica (son únicamente sistematizables para tipos particulares de gramáticas, requieren transformaciones laboriosas y propensas a errores, así como soluciones *ad hoc* específicas para gramática particular en el caso de tipos de gramáticas más generales). De esta forma, durante esta tarea se pondrá especial énfasis en obtener una estrategia de codificación que, por una parte, sea capaz de sistematizar la codificación de gramáticas de atributos no circulares arbitrarias, y que, por otra parte, resulte natural e intuitiva para los alumnos.

El principal objetivo de la estrategia propuesta será que los alumnos entiendan la relación existente entre el procesador de lenguaje especificado y la implementación obtenida. Para ello, la estrategia implicará directamente al

estudiante en el proceso de codificación, para que este sea consciente de las relaciones existentes entre la gramática de atributo de partida y la implementación resultante de la aplicación del enfoque de codificación planteado.

3.2.2.2 Desarrollo del software de soporte

La estrategia de codificación de gramáticas de atributos como esquemas de traducción planteada deberá apoyarse en herramientas de generación de procesadores de lenguaje basadas en esquemas de traducción, por lo que será necesario desarrollar una biblioteca software que facilite la realización de la evaluación semántica determinada por las gramáticas de atributos en los esquemas de traducción codificados. Dicha biblioteca facilitará la aplicación de los patrones de codificación, y permitirá que los estudiantes serán capaces de identificar, tanto sintáctica como semánticamente, la relación existente entre la implementación obtenida en forma de esquema de traducción y la gramática de atributos de partida.

3.2.2.3 Validación de la estrategia y del software de soporte

Una vez formulada la estrategia e implementado el software de soporte a la misma, será necesario validar la utilidad educativa, efectiva y percibida, por docentes y estudiantes. Para ello, se seguirá un procedimiento análogo al planteado en relación con el primer objetivo:

- Será necesario plantear experiencias con docentes especialistas en la enseñanza de la materia de Procesadores de Lenguaje para obtener la utilidad educativa percibida por los profesores, así como la idoneidad de la estrategia dentro de un curso típico de este tipo de materias.
- Por otro lado, se plantearán experiencias con estudiantes para observar la eficacia educativa del método de desarrollo ideado, frente a otros enfoques de desarrollo. También, durante estas experiencias, se recabará la opinión de los estudiantes sobre el enfoque de desarrollo propuesto, así como la utilidad educativa percibida por estos.

Los resultados que se obtendrán de docentes y estudiantes proporcionarán suficiente información para comprobar la eficacia educativa del enfoque, así como la utilidad percibida y valoración por parte de los principales actores implicados en el enfoque. De esta forma, se podrá comprobar la validez del enfoque y su idoneidad para su uso en los cursos de Procesadores de Lenguaje, así como plantear de manera dirigida posibles mejoras al mismo.

3.2.3 Actividades relativas a la creación del entorno educativo de desarrollo de procesadores de lenguaje basado en gramáticas de atributos

Para cumplir el objetivo 3 descrito anteriormente será necesario, en primer lugar, construir una plataforma software para el desarrollo de procesadores de lenguaje especificados mediante el formalismo de las gramáticas de atributos que: (i) permita a los alumnos entender el funcionamiento de sus propias especificaciones, basadas en dicho formalismo, apoyándose en la estrategia

planteada en el objetivo 1, y (ii) que genere implementaciones de los procesadores especificados basándose en la estrategia de desarrollo planteada en el objetivo 2. Así mismo, será necesario validar el sistema software desarrollado y su aplicación en un curso de Procesadores de Lenguaje. Para ello, se organizarán diferentes experiencias entre docentes y alumnos de dichas materias.

3.2.3.1 Desarrollo de la plataforma software

Para alcanzar el objetivo 3 planteado en la sección anterior, es necesario diseñar, y desarrollar, un entorno de desarrollo de procesadores de lenguaje basados en su especificación mediante el formalismo de las gramáticas de atributos, que conjugue los dos enfoques planteados en los objetivos 1 y 2: facilitar la comprensión de las especificaciones y de la subsiguiente implementación como esquema de traducción. Para ello:

- La plataforma proporcionará herramientas que faciliten la comprensión de conceptos básicos de las gramáticas de atributos especificadas por el alumno mediante un potente depurador visual basado en la estrategia educativa formulada en el objetivo 1.
- La plataforma generará, a su vez, procesadores de lenguaje basados en los mecanismos de codificación mediantes esquemas de traducción, utilizando la estrategia de codificación formulada en el objetivo 2. Dicha estrategia de codificación se llevará a cabo de forma automática, por la herramienta, permitiendo a los alumnos explorar el esquema de traducción codificado y encontrar la relación existente con la correspondiente gramática de atributos introducida como especificación de partida.

Es importante remarcar, por último, que dicha plataforma, al contrario que otras plataformas orientadas a desarrolladores más experimentados (por ejemplo, LISA), tendrá un carácter fundamentalmente educativo, y totalmente conectado con las estrategias desarrolladas en los dos primeros objetivos de esta tesis.

3.2.3.2 Validación del sistema

Cuando el desarrollo de la plataforma se haya completado, será necesario validar el software creado. Como en los casos anteriores, se organizarán diferentes experiencias entre docentes y alumnos de los cursos de Procesadores de Lenguaje para recabar datos sobre la utilidad educativa percibida de la herramienta. Con estos datos, se podrá validar la utilidad de la plataforma, así como obtener información útil para su mejora.

3.3 A modo de conclusión

A fin de promover la mejora de la adquisición de los conceptos y técnicas esenciales para la especificación e implementación sistemática dirigida por especificaciones de procesadores de lenguaje, en esta tesis se han planteado objetivos orientados a:

- La formulación de una estrategia para facilitar la comprensión de conceptos básicos del formalismo de las gramáticas de atributos, apoyado por un sistema software que haga uso de las nuevas tendencias en eLearning y educación.
- La formulación de un método de desarrollo y codificación de gramáticas de atributo no circulares como esquemas de traducción, cuyo objetivo es facilitar la aplicación de las técnicas sistemáticas de desarrollo de procesadores de lenguaje a partir de su especificación.
- La creación de una plataforma de desarrollo que conjugue los enfoques implícitos en los dos objetivos anteriores para que los estudiantes sean capaces de comprender sus propias especificaciones dadas en forma de gramática de atributos y sus subsecuentes implementaciones en forma de esquemas de traducción.

De esta forma, en el siguiente capítulo se presenta una discusión del contenido de los artículos que acompañan a la presente tesis con el fin de integrar sus contenidos y relacionarlos con los objetivos presentados en el presente capítulo. Por otro lado, en el Capítulo 5 se analiza en qué medida se han cumplido los objetivos planteados respecto a los resultados presentados en los artículos y, también se describen las principales líneas de trabajo futuro.

Capítulo 4: Discusión de las contribuciones de los artículos

Este capítulo contextualiza los resultados de investigación obtenidos durante el desarrollo de esta tesis en base a las publicaciones editadas que la soportan. La sección 4.1 contextualiza los resultados relativos a la estrategia educativa y software de soporte para mejorar la comprensión de conceptos básicos sobre gramáticas de atributos (objetivo 1 de la tesis). La sección 4.2 contextualiza los resultados relativos a la codificación de gramáticas de atributos no circulares arbitrarias sobre esquemas de traducción (objetivo 2). La sección 4.3 resume, por último, los resultados relativos a la construcción de herramientas de desarrollo de procesadores de lenguaje basadas en gramáticas de atributos que integran las citadas estrategias de comprensión y codificación de gramáticas (objetivo 3).

4.1 Estrategia para la comprensión de las especificaciones basadas en gramáticas de atributos

La estrategia de enseñanza ideada para asistir a docentes y alumnos en el proceso de aprendizaje/enseñanza de conceptos básicos del formalismo de las gramáticas de atributo se apoya en un sistema software que permite a los docentes proporcionar baterías de ejercicios basados en problemas de procesamiento de lenguaje donde los estudiantes deben: analizar y comprender el lenguaje propuesto, realizar el procesamiento de la sentencia proporcionada, y demostrar que son capaces de emular el modelo computacional del formalismo adecuadamente. Con el propósito de motivar a los estudiantes, se decidió presentar estas baterías de ejercicios mediante simuladores interactivos que aprovechen, para dicho propósito, las nuevas tendencias en eLearning.

La subsección 4.1.1 describe la estrategia educativa planteada para facilitar la adquisición de conceptos básicos de las gramáticas de atributos. La subsección 4.1.2 describe el sistema software desarrollado para dar soporte a dicha estrategia. La subsección 4.1.3 describe las diferentes experiencias entre docentes y estudiantes realizadas para validar la estrategia educativa y el sistema software que le da soporte. Por último, la subsección 4.1.4 plantea un modelo de proceso para guiar el desarrollo de sistemas software educativos basados en enfoques generativos similares al planteado en la subsección 4.1.2.

4.1.1 Formulación de la estrategia

La estrategia educativa planteada para facilitar el aprendizaje de los conceptos relativos al formalismo de las gramáticas de atributos en etapas tempranas del aprendizaje se centra, principalmente, en los aspectos relativos a la evaluación semántica de dicho formalismo. Tal y como se describe en [135] [138] [137] [139] [141] [142], la estrategia educativa aboga por la provisión de baterías de ejercicios en los cuales se proponen tareas de procesamiento de lenguaje. Estas tareas estarán descritas por un lenguaje, especificado mediante una gramática de atributos, una sentencia para dicho lenguaje y el árbol sintáctico derivado del análisis de la sentencia. El objetivo de los alumnos es proporcionar un orden de evaluación de cada una de las instancias de atributo que decoran el árbol sintáctico, así como su valor. De esta forma, los estudiantes son capaces de entender cómo se realiza la evaluación semántica de acuerdo con el formalismo.

4.1.2 Desarrollo del sistema software de soporte a la estrategia

Con el fin de soportar la estrategia educativa planteada en la subsección anterior, se ha llevado a cabo el desarrollo de un sistema software para: (i) la creación de baterías de ejercicios del tipo de los descritos anteriormente, (ii) la generación de simuladores interactivos basados en dichos ejercicios y (iii) la evaluación del progreso de los estudiantes. En [135] [138] [137] [141] [142] se describen las diferentes herramientas software que componen este sistema educativo, así como la evolución de dichas herramientas. Este conjunto de herramientas, que es consecuencia directa de los requisitos planteados en el punto 3.2.1.2 para cumplir con el objetivo 1 de la tesis, consta de una *herramienta de autoría*, *generadores de simuladores* y una *herramienta de análisis*.

La *herramienta de autoría* proporciona la suficiente expresividad para describir los tipos de ejercicios que los estudiantes resolverán, utilizando los simuladores generados, y que han sido descritos en la subsección 4.1.1. Inicialmente, y tal y como se describe en [135] [138] [137] [142], se construyó una herramienta que permitía editar directamente las posibles soluciones a los ejercicios. Para ello, la herramienta permitía editar el árbol de análisis sintáctico para la sentencia de entrada, así como el grafo de dependencias entre atributos. Seguidamente, y tal y como se describe en [135] [138] [141], se construyó una herramienta más usable, que proporciona a los docentes un editor de gramáticas de atributos donde estos sólo tienen que proporcionar la especificación de la gramática de atributos y su sentencia. Con esta información la herramienta genera automáticamente las posibles soluciones (el árbol sintáctico y el grafo de dependencias, necesario para

comprobar los órdenes de evaluación semánticos proporcionados por los estudiantes).

Los *generadores de simuladores* son capaces de generar los diferentes simuladores a partir de baterías de ejercicios creados mediante la herramienta de autoría. Durante el desarrollo de esta tesis se han construido dos generadores diferentes:

- Un generador que genera automáticamente juegos serios a partir de las baterías de ejercicios. Los juegos serios generados están basados en un videojuego de tipo puzle. El videojuego presenta al usuario un laberinto, que representa el árbol de análisis sintáctico, donde encontrará habitaciones, que representan los nodos del árbol, y pasillos que las conectan, relacionados con los arcos del árbol. Dentro de estas habitaciones hay mesas con cajas que representan los atributos de cada nodo. Estas cajas pueden contener objetos, que se corresponden con los valores de dichos atributos. El objetivo del usuario es mover estos objetos de unas cajas a otras, dirigido por las dependencias entre atributos especificadas en las gramáticas de los ejercicios, con el fin de activar el valor de cada una de las cajas del laberinto. Así, el estudiante proporciona, implícitamente, el orden de evaluación de los atributos del árbol. El generador y el tipo de juegos generados se describen en [135] [138] [137] [141] [142].
- Un segundo generador que genera simulaciones interactivas. Los simuladores generados presentan toda la información contenida en cada ejercicio de la batería de una forma atractiva para el alumno. El lenguaje, descrito mediante una gramática de atributos, y la sentencia a procesar se presentan al alumno de forma textual. El árbol sintáctico queda representado de forma gráfica formando el eje central del simulador. Los atributos de los nodos se representan mediante cuadrados, que aparecen al seleccionar un nodo del árbol sintáctico. Por último, el alumno puede consultar información sobre los atributos seleccionándolos. La mecánica del simulador es idéntica a la de los juegos serios, aunque evitando el componente más lúdico, que puede distraer a los alumnos del objetivo final de emulación del proceso de evaluación semántica. De esta forma, el alumno debe explorar los nodos del árbol sintáctico en busca de aquellos atributos cuyo valor no haya sido calculado y, guiado por las dependencias descritas en la gramática de atributos, mover los valores necesarios a aquellos atributos que lo necesiten para calcular su valor. Además, por cada una de las acciones de cálculo realizadas por el alumno, la herramienta proporciona información de la validez o invalidez de estas, para orientar al alumno en la resolución del ejercicio. Este generador y el tipo de simuladores generados se describen en [135] [138] [139].

Finalmente, la *herramienta de análisis* proporciona a los docentes un entorno para estudiar y evaluar las acciones llevadas a cabo por los estudiantes durante la resolución de las baterías de ejercicios en los simuladores. Durante la tesis, y tal y como se describe en [135] [138], se construyó, primeramente una herramienta de análisis simple, que mostraba de forma textual las acciones realizadas por los alumnos. Seguidamente dicha herramienta evolucionó a una que presenta de forma visual, sobre una representación del árbol sintáctico, las acciones realizadas

por el estudiante durante el proceso de resolución de cada problema. Dicha herramienta se describe en [135] [138] [141].

Las herramientas descritas anteriormente componen el sistema educativo *Evaluators*, que es capaz de generar tanto juegos educativos como simuladores interactivos basados en visualizaciones a partir de baterías de ejercicios orientados a la enseñanza de conceptos básicos de las gramáticas de atributos. Tal y como se describe en [135] [138], los generadores funcionan sobre las mismas baterías de ejercicios. Por tanto, los docentes pueden generar juegos serios o simulaciones interactivas a partir de una fuente única de ejercicios. Así mismo, las herramientas de análisis también funcionan indistintamente sobre las salidas proporcionadas por los dos tipos de simuladores (juegos serios, y simulaciones interactivas). El sistema en sí da soporte software a la estrategia educativa planteada en la sección anterior, proporcionando herramientas para la creación de los ejercicios, herramientas para generar los simuladores, y herramientas para evaluar las soluciones dadas por los alumnos.

4.1.3 Validación de la estrategia y del software asociado

Con el fin de validar tanto la estrategia educativa, como el software educativo que le da soporte, durante los años académicos del 2010-2011, 2011-2012 y 2012-2013 ambos empezaron a implantarse en la Facultad de Informática de la Universidad Complutense de Madrid, de forma experimental. Tanto para la estrategia educativa, como para el sistema software que le da soporte, se realizaron estudios que involucraron tanto a docentes como a estudiantes. De esta forma, en relación con los estudios con docentes:

- Durante el curso académico del 2010-2011 se realizó un estudio con docentes del curso de Procesadores de Lenguaje con el fin de evaluar la versión inicial de la herramienta de autoría, basada en un enfoque constructivista, del sistema *Evaluators*. Los resultados demostraron que la herramienta de autoría inicial no era adecuada para el grupo de docentes entrevistados y fueron el germen de la versión actual, basada en un enfoque generativo, de la herramienta de autoría [135] [142]. Posteriormente se repitió este estudio a mediados del curso académico 2010-2011 de valoración con la nueva versión de la herramienta, donde se obtuvieron resultados positivos [135].
- También durante el curso 2010-2011 se llevó a cabo otro estudio con docentes del curso para evaluar la herramienta de análisis de *Evaluators*. Las observaciones obtenidas no fueron positivas, lo que derivó en un replanteamiento de la herramienta que desembocó en la versión actual de la herramienta [135] [142], cuya evaluación posterior (durante el curso académico del 2011-2012) fue positiva [135].

Por su parte, en lo que se refiere a los estudios llevados a cabo con estudiantes:

- Durante el curso académico del 2010-2011 se realizó una experiencia con estudiantes donde se evaluó la eficacia educativa de la estrategia a corto plazo y el software basado en juegos serios, así como el grado de satisfacción de los estudiantes. En dicha experiencia, quedó patente que la eficacia educativa era similar a la obtenida con los métodos de enseñanza

tradicionales, pero el grado de satisfacción era mayor, lo que aumentaba la motivación de los estudiantes a estudiar las materias tratadas más activamente que con el método tradicional [135] [137].

- Durante el curso académico del 2011-2012 se repitieron las experiencias orientadas a evaluar tanto el grado de satisfacción de los estudiantes, como la eficacia educativa a corto plazo de la estrategia y el sistema basado en la generación de juegos serios. En dicho estudio se obtuvieron resultados similares a los obtenidos durante el año académico 2010-2011 [135].
- Aparte de llevar a cabo experiencias de evaluación a corto plazo, durante los cursos 2010-2011 y 2011-2012 todos los estudiantes siguieron la estrategia y tuvieron ocasión de utilizar el software de soporte. Esto nos permitió realizar un estudio a largo plazo, comparando los resultados obtenidos durante estos dos años y los años anteriores. Este estudio puso de manifiesto que el uso de la estrategia y el sistema basado en juegos serios tuvo un efecto positivo en el aprendizaje de los estudiantes respecto a otros años académicos donde no se emplearon ninguno de ellos [135] [141].
- Finalmente, durante el curso académico del 2012-2013 se realizó una experiencia orientada a medir el grado de satisfacción de los estudiantes, así como la eficacia educativa de la estrategia y el software de soporte que genera simuladores basados en visualizaciones interactivas. Los estudiantes mostraron un alto grado de satisfacción con el uso de los simuladores, considerándolos adecuados para el aprendizaje. Además, el estudio comparativo para medir la eficacia educativa mostró resultados muy prometedores, respecto al método de enseñanza tradicional de los conceptos tratados [135] [139].

Así pues, los resultados obtenidos de docentes y estudiantes ponen de manifiesto que tanto la estrategia educativa, como el sistema software que la acompaña (con los dos posibles tipos de simuladores generados), son adecuados para la enseñanza de conceptos relativos al cálculo semántico derivado del formalismo de las gramáticas de atributos. Además, también se observa una acogida positiva de ambos, estrategia educativa y sistema software, por parte de docentes y alumnos.

4.1.4 Formulación del modelo de proceso

Para finalizar la consecución del objetivo 1, en esta tesis se ha abstraído el modelo de desarrollo subyacente a *Evaluators* para facilitar la construcción de sistemas similares. El modelo de proceso resultante proporciona, de cara al objetivo general de esta tesis, una metodología de trabajo para producir sistemas software similares a *Evaluators*, pero centrados en otros formalismos de especificación de procesadores de lenguaje.

El modelo de proceso constituye una estrategia para el desarrollo de sistemas de construcción de simulaciones educativas basados en un enfoque generativo. De esta forma, los sistemas desarrollados son capaces de generar de forma automática simulaciones educativas a partir de colecciones de ejercicios. El modelo organiza la labor de docentes, desarrolladores y estudiantes en diferentes tareas. Las primeras tareas propuestas por el enfoque se centran en establecer los ejercicios y el tipo de

simulación educativa que docentes y desarrolladores desean contemplar para el sistema. Después, los desarrolladores crean las diferentes herramientas software que componen el sistema a partir de las especificaciones acordadas en la tarea anterior: la herramienta de autoría, el generador de simuladores y la herramienta de análisis. En las posteriores tareas, los docentes emplean las herramientas desarrolladas para crear baterías de ejercicios (mediante la herramienta de autoría) y los correspondientes simuladores (utilizando el generador de simuladores). A continuación, los estudiantes utilizan los simuladores educativos generados para resolver los problemas propuestos por los docentes. Los simuladores generados serán capaces de almacenar la actividad de los estudiantes en ficheros de registro, que posteriormente los docentes podrán analizar (utilizando la herramienta de análisis) para evaluar las soluciones de los estudiantes.

En paralelo al proceso de producción explicado anteriormente, existe otro proceso de evaluación de los productos generados que completa la estrategia de desarrollo. El objetivo de estas evaluaciones es identificar problemas y aspectos de mejora de los productos software creados para que se adapten a las necesidades de los usuarios finales de estos productos, estudiantes y docentes. Así pues, cuando un producto es creado durante el flujo principal de trabajo de la estrategia propuesta, se crea a su vez un *instrumento de evaluación* donde se especifican los aspectos claves a evaluar del producto recién creado y qué método de evaluación será empleado. Posteriormente, cuando este producto se utilice, se aplicará a su vez el instrumento de evaluación asociado a él para que sea evaluado por el actor implicado (estudiante o docente), registrando en un *informe de evaluación* los resultados de valoración obtenidos. Estos informes son analizados y pueden provocar reactivaciones de las diferentes tareas del flujo principal de producción. Estas reactivaciones están orientadas a mejorar productos previamente creados con la información recogida en el *informe de evaluación*.

En [137] se describe una versión preliminar del modelo de proceso enfocada en el desarrollo de juegos serios. En [138] se describe la versión actual del mismo, focalizada en la generación de simulaciones interactivas en general. En [135] se detalla cómo dicho modelo de proceso emerge de la experiencia de desarrollo de *Evaluators*.

4.1.5 Conclusiones

A lo largo de esta sección se han contextualizado, en base a las publicaciones, las diferentes tareas acometidas para cumplir con el *objetivo 1*, es decir, para proporcionar, soportar mediante herramientas software y evaluar una estrategia educativa para mejorar la enseñanza/aprendizaje de conceptos básicos de las gramáticas de atributos en etapas tempranas de la formación. De esta forma, en primer lugar se concibió la estrategia educativa como una centrada en el aspecto más problemático del formalismo: su modelo de cómputo no convencional, dirigido por las dependencias entre atributos. Para ello se adoptó un enfoque dirigido por problemas, diseñando un tipo específico de ejercicio orientado a trabajar dicho aspecto mediante la experimentación. Como consecuencia, la estrategia se pudo soportar de manera natural mediante la generación de juegos serios y simulaciones interactivas en *Evaluators*.

Las experiencias de evaluación, llevadas a cabo con docentes y estudiantes para validar la adecuación y la efectividad educativa de la estrategia y su correspondiente software de apoyo, han puesto de manifiesto la buena acogida entre docentes y estudiantes de dicha estrategia y software de apoyo frente a métodos de enseñanza más tradicionales. Estas experiencias también han evidenciado mejoras en lo referente a la eficacia educativa de los métodos empleados.

Finalmente, el enfoque seguido durante la construcción del sistema software se ha podido también abstraer de manera satisfactoria mediante un modelo de proceso aplicable a otros escenarios. El modelo de proceso resultante puede utilizarse para dar soporte a estrategias dirigidas por problemas a través de un enfoque generativo modulado por las valoraciones de docentes y estudiantes. En el marco de esta tesis, dicho modelo permitirá extrapolar las propuestas realizadas a otros escenarios que involucren formalismos diferentes, e incluso a otros contextos de enseñanza/aprendizaje.

4.2 Estrategia para la implementación de gramáticas de atributos como esquemas de traducción

La estrategia de desarrollo ideada para facilitar el desarrollo de los procesadores de lenguaje de los alumnos permite a estos obtener implementaciones funcionales, como esquemas de traducción, a partir de especificaciones basadas en gramáticas de atributos no circulares arbitrarias. Para ello, dicha estrategia se apoya en el uso de una biblioteca software que facilita dicha tarea. Finalmente, con el objetivo de validar tanto la estrategia de desarrollo como el software de soporte, se han llevado a cabo diferentes estudios y experiencias realizadas en un curso de Procesadores de Lenguaje.

La subsección 4.2.1 contextualiza la estrategia de desarrollo. La subsección 4.2.2 contextualiza el desarrollo de la biblioteca. La subsección 4.2.3, por último, contextualiza la evaluación realizada.

4.2.1 Formulación de la estrategia de desarrollo

En [143] [144] se describe la estrategia de desarrollo formulada. Dicha estrategia es capaz de obtener la implementación de un procesador de lenguaje, mediante el uso de herramientas de generación de procesadores basadas en esquemas de traducción, a partir de su especificación como una gramática de atributos. En concreto, la estrategia se basa en los siguientes procesos para obtener la implementación del procesador del lenguaje especificado:

- En primer lugar, las reglas sintácticas de la especificación de entrada, descritas en términos de una gramática de atributos, se transcriben en términos procesable por una herramienta de generación basada en esquemas de traducción. En concreto, en [144] la estrategia desarrollada aboga por la utilización de herramientas de generación de analizadores/traductores *bottom-up*, como CUP, aunque en [143] la

estrategia se generaliza también a generadores de analizadores/traductores descendentes, como ANTLR o JavaCC.

- Posteriormente, se deben transcribir las ecuaciones semánticas como acciones semánticas dentro del esquema de traducción. Para ello, se deben describir las ecuaciones semánticas en términos de las dependencias entre atributos que se derivan de ellas, y del método de cálculo necesario.

Los traductores resultantes acoplan el proceso de evaluación semántica con el proceso de análisis. Así mismo, dicho proceso de evaluación semántica puede estar dirigido por las dependencias, o bien proceder bajo demanda. En cualquier caso, todos estos son aspectos transparentes para el alumno que realiza la codificación. De esta forma, con la estrategia presentada, los alumnos son capaces de obtener implementaciones de los procesadores de lenguajes que han diseñado directamente desde su especificación en términos de una gramática de atributos no circular arbitraria. Aparte de ser capaces de validar su diseño de una forma experimental, los estudiantes son capaces de identificar la correspondencia entre el esquema de traducción y la especificación de partida descrita mediante el formalismo de las gramáticas de atributos. Así mismo, como se describe en [143] [144], los alumnos también pueden refinar la implementación a través de sucesivas optimizaciones, pudiendo trazar siempre las relaciones existentes con la especificación original.

4.2.2 Desarrollo del software de soporte

La estrategia de desarrollo propuesta anteriormente requiere de un software de soporte. En concreto, y tal y como se describe en [136], se ha desarrollado una biblioteca software, llamada *EvLib*, que proporciona los métodos necesarios para especificar las ecuaciones semánticas como acciones semánticas dentro del esquema de traducción obtenido de la aplicación directa de la estrategia de desarrollo. En dichas acciones semánticas será necesario especificar, utilizando la biblioteca, las dependencias entre atributos y el método de cálculo. Dichas dependencias y método de cálculo, se derivan de la ecuación semántica correspondiente. Además, *EvLib* proporciona un motor de evaluación semántica para que los procesadores generados sean capaces de realizar los cálculos semánticos adecuadamente.

4.2.3 Validación de la estrategia de desarrollo y del software de soporte

Para validar la adecuación de la estrategia y el software de soporte, se organizaron diferentes experiencias entre docentes y estudiantes del curso 2012-2013 de Procesadores de Lenguaje en la Universidad Complutense de Madrid. Estos resultados se encuentran en los trabajos descritos en [136]. En primer lugar, se muestran los resultados de valoración obtenidos de docentes y estudiantes, donde valoran el enfoque positivamente. A pesar de que la estrategia es considerada por los estudiantes como adecuada para desarrollar sus procesadores, consideran insuficiente o confusa la información de depuración proporcionada por las herramientas de generación de procesadores y la biblioteca *EvLib*. Por otro lado, en [136] también se registran los resultados de la eficacia educativa respecto a las estrategias de implementación tradicionales empleadas en el curso. Se puede

observar que los estudiantes que emplearon la estrategia de desarrollo que preserva el diseño original han obtenido mejores resultados que los alumnos que emplearon la estrategia tradicional.

Con estos resultados, se puede concluir que la estrategia de desarrollo, así como el software educativo, son adecuados para los estudiantes de un curso de Procesadores de Lenguaje, quedando ambos aspectos validados para su propósito: facilitar el desarrollo de los procesadores de lenguaje de los alumnos.

4.2.4 Conclusiones

A modo de conclusión, la estrategia de desarrollo propuesta, y apoyada por la biblioteca *EvLib*, proporciona a los estudiantes un marco que les permite desarrollar sus propios procesadores a partir de la especificación en términos de una gramática de atributos sin necesidad de utilizar herramientas especializadas (únicamente las herramientas convencionales de desarrollo basadas en esquemas de traducción). Con el enfoque de desarrollo propuesto los alumnos cuentan con herramientas para desarrollar (y validar) sus procesadores de lenguaje directamente desde su especificación y encontrar una correspondencia directa entre dicha especificación y su implementación subsiguiente, tal como demuestran los resultados obtenidos en las diferentes experiencias presentadas en [136]. Así mismo, tal y como se discute en [143] [144], el enfoque permite también una estrategia basada en evolución de prototipos, donde el prototipo inicial obtenido puede transformarse en una implementación final mediante la aplicación sucesiva de refinamientos, pudiendo siempre relacionarse los artefactos intermedios con la especificación original.

4.3 Creación del entorno educativo de desarrollo de procesadores de lenguaje basado en gramáticas de atributos

El objetivo 3 planteado en esta tesis se ha satisfecho construyendo una plataforma para el desarrollo de procesadores de lenguaje basada en el formalismo de las gramáticas de atributos que facilita la comprensión de este formalismo mediante el uso de las estrategias empleadas para conseguir el objetivo 1, y que, a su vez, genera implementaciones de los procesadores basándose en los procesos propuestos para cumplir el objetivo 2. Además, se ha llevado a cabo una validación empírica de la adecuación y utilidad educativa del sistema desarrollado para su uso en un curso de Procesadores de Lenguaje.

La subsección 4.3.1 contextualiza los resultados relativos al desarrollo de la plataforma. La subsección 4.3.2 contextualiza los resultados relativos a su evaluación.

4.3.1 Desarrollo de la plataforma software

La plataforma de desarrollo creada, y descrita en [140], está destinada a la generación automática de procesadores de lenguaje a partir de su especificación

en términos de una gramática de atributos. La herramienta proporciona un editor de gramáticas de atributos completo para permitir a los estudiantes especificar sus procesadores de lenguaje mediante dicho formalismo. Por otro lado, la herramienta cuenta con un potente depurador visual que, dado un procesador especificado y una sentencia para dicho procesador, muestra el correspondiente árbol de análisis sintáctico decorado y emula el proceso de cómputo de cada uno de los valores de las instancias de atributo que decoran dicho árbol. Dicho depurador se apoya en la estrategia de comprensión desarrollada para satisfacer el objetivo 1. De esta forma, los estudiantes serán capaces de tener una mayor comprensión del funcionamiento interno de sus propios procesadores de lenguaje. Por último, la plataforma genera los procesadores de lenguaje a partir de su especificación en forma de gramática de atributos aplicando de forma automática la estrategia de desarrollo planteada para cumplir el objetivo 2. De esta forma, las implementaciones generadas mantendrán una correspondencia legible por los alumnos con la especificación de partida.

4.3.2 Validación del sistema

La validación del sistema se llevó a cabo durante una estancia de investigación en la Facultad de Informática de la Universidade do Minho (Portugal), en el grupo del Prof. Pedro Rangel Henriques. Dicha estancia se realizó en el curso 2013-2014, y permitió llevar a cabo una experiencia con alumnos del curso de Procesadores de Lenguaje de la citada Facultad con el fin de medir el grado de satisfacción de estos con el entorno y validar la utilidad educativa percibida. En [140] se registran los resultados obtenidos, donde se puede observar que la herramienta ha obtenido éxito entre los alumnos, quienes la consideran útil a la hora de especificar, implementar y depurar sus propios procesadores de lenguaje. A pesar de las posibles mejoras y problemas que los estudiantes encontraron durante la experiencia realizada, tanto el editor de gramáticas de atributos como el depurador visual fueron valorados positivamente.

4.3.3 Conclusiones

A modo de conclusión, el entorno de desarrollo creado permite satisfacer el objetivo 3 de proporcionar herramientas a los alumnos para facilitar el desarrollo de procesadores de lenguaje a partir de su especificación. El entorno de desarrollo creado proporciona las herramientas necesarias para que los alumnos sean capaces de entender y refinar sus propias especificaciones como gramáticas de atributos gracias a su editor y al depurador visual. Además, la herramienta automatiza los pasos de implementación que el alumno debería abordar para obtener el traductor desde su especificación siguiendo la estrategia de desarrollo planteada en la sección anterior, permitiendo a los alumnos identificar la correspondencia existente entre la especificación introducida y la implementación obtenida.

Capítulo 5: Conclusiones y trabajo futuro

En los capítulos anteriores se han presentado diferentes problemas y necesidades educativas en el ámbito de la enseñanza de Procesadores de Lenguaje. Para dichas necesidades se han propuesto diferentes estrategias y herramientas software cuyo principal objetivo es mejorar la experiencia educativa de docentes y estudiantes relativa a conceptos de un curso de Procesadores de Lenguaje, y, más concretamente, relativa a conceptos relacionados con el formalismo de especificación de las gramáticas de atributos. Tal y como se ha detallado en el capítulo anterior, los resultados obtenidos a la hora de cubrir los objetivos propuestos en este trabajo de tesis se detallan en diferentes publicaciones que integran esta memoria. De este modo, este último capítulo concluye esta memoria de tesis resumiendo las conclusiones principales y presentando algunas líneas de trabajo futuro.

5.1 Principales aportaciones

En esta sección se presentan las principales aportaciones realizadas en este trabajo de tesis. Más concretamente, y en base a la discusión mantenida en los capítulos precedentes, se destacan las siguientes aportaciones:

- Propuesta de una estrategia educativa para la comprensión de las especificaciones basadas en gramáticas de atributos y soportada por un sistema software capaz de generar distintos tipos de simuladores educativos.
- Formulación de una estrategia para la implementación de gramáticas de atributos no circulares arbitrarias como esquemas de traducción.
- Creación de un entorno de desarrollo de procesadores de lenguaje basados en su especificación mediante gramáticas de atributos que facilita la

comprensión de las especificaciones y su implementación como esquemas de traducción.

A continuación, se describen en detalle cada una de estas aportaciones.

5.1.1 Estrategia educativa para la comprensión de las especificaciones basadas en gramáticas y software de generativo de soporte

En este trabajo de tesis se ha formulado una estrategia educativa para mejorar el proceso de enseñanza/aprendizaje de conceptos básicos del formalismo de las gramáticas de atributos en etapas tempranas del aprendizaje dentro de un curso de Procesadores de Lenguaje. Dicha estrategia se centra en enseñar conceptos relativos al cálculo semántico derivado del formalismo mediante el uso de baterías de ejercicios relativos a dichos conceptos. Dichos ejercicios propondrán a los estudiantes un problema de procesamiento de un lenguaje proporcionando la siguiente información:

- Descripción informal de un procesador de lenguaje.
- Descripción formal del mismo procesador de lenguaje especificada mediante el formalismo de las gramáticas de atributos.
- Una sentencia del lenguaje a procesar.
- El árbol sintáctico decorado derivado de realizar el procesamiento de la sentencia con la gramática de atributos proporcionada.

Con esta información, la solución que los estudiantes deberán proporcionar será un orden de evaluación correcto, acorde al lenguaje definido. Cabe destacar que pueden existir diferentes órdenes de evaluación válidos para un mismo ejercicio. Además, para dar soporte a la estrategia, se ha creado un sistema software, llamado *Evaluators*, que se caracteriza por:

- Proporcionar herramientas de autoría para facilitar la construcción de las baterías de ejercicios a los docentes y que permite a estos introducir la descripción del lenguaje y la sentencia para generar toda la información necesaria para definir cada ejercicio de la batería.
- Generadores que son capaces de construir simuladores educativos basados en juegos serios y en visualizaciones interactivas a partir de dichas baterías de ejercicios. Será en estos simuladores generados donde los estudiantes resolverán los ejercicios propuestos por los docentes. Ambos tipos de simuladores presentan la información de cada ejercicio de la siguiente forma:
    - Los simuladores basados en juegos serios presentan al estudiante un laberinto, basado en la estructura del árbol sintáctico, donde encontrará cajas, que representan los atributos. El objetivo es mover los objetos, que representan el valor semántico de cada atributo, de unas cajas a otras con el fin de crear nuevos objetos. De esta forma, el estudiante proporcionará implícitamente un orden de evaluación de los atributos.
    - Los simuladores basados en visualizaciones interactivas tienen la misma dinámica que los simuladores basados en juegos serios pero presentan la información mediante representaciones visuales muy cercanas a las representaciones formales usadas en clase. El

simulador muestra una representación del árbol sintáctico decorado y sobre esta los estudiantes deben mover los valores de los atributos, representados por cajas, para construir el valor de los atributos pendientes de computación. Y en este proceso, los alumnos especifican un orden de evaluación.

- Proporcionar herramientas de análisis donde los docentes podrán valorar las soluciones propuestas por los estudiantes en los simuladores generados y controlar su avance mostrando sobre una representación de cada árbol sintáctico de la batería de ejercicios las acciones llevadas a cabo por los estudiantes durante la simulación.

Por otro lado, y con el fin de validar tanto la estrategia educativa como el sistema software *Evaluators* que le da soporte, se han llevado diferentes experiencias entre docentes y estudiantes del curso de Procesadores de Lenguaje. En estas experiencias ha quedado patente cómo las valoraciones recogidas entre docentes y estudiantes son buenas, tanto las relativas a la estrategia formulada como al software que le da soporte. Además, los estudios descritos en el capítulo anterior relativos a la eficacia educativa de la estrategia y el software también han demostrado la utilidad de éstos para los estudiantes y su aprendizaje, ya sea a corto plazo (con experiencias puntuales) como a largo plazo (con un uso continuo a lo largo de diferentes cursos de Procesadores de Lenguaje).

Por último, toda la experiencia ganada durante el desarrollo de la estrategia educativa y el sistema software que le da soporte ha sido resumida en un modelo de proceso para la producción de sistemas educativos generativos basados en baterías de ejercicios y dirigido por las evaluaciones de docentes y estudiantes. Dicho modelo de proceso tiene como principal objetivo abstraer las buenas prácticas observadas durante el desarrollo de *Evaluators* y poder aplicarlas a otros ámbitos educativos, en general, y en concreto, en el curso de Procesadores de Lenguaje, para otros formalismos de especificación.

5.1.2 Estrategia para la implementación de gramáticas de atributos como esquemas de traducción

Con el fin de ayudar a los estudiantes de un curso de Procesadores de Lenguaje a identificar aspectos relativos a sus propias especificaciones en las implementaciones que deben realizar como parte de las diferentes tareas que se les proponen a lo largo del curso, en este trabajo de tesis se ha propuesto una estrategia de desarrollo de procesadores de lenguaje como esquemas de traducción a partir de gramáticas de atributos no circulares arbitrarias. Dicha estrategia permite a los estudiantes producir esquemas de traducción que preservan la especificación de partida, en forma de gramática de atributos, lo que facilita a los estudiantes depurar posibles errores, así como comprender la relación existente entre la especificación de procesadores mediante gramáticas de atributos y su implementación final al haber una correspondencia directa entre la especificación de partida y la implementación producida mediante dicha estrategia.

La estrategia educativa de desarrollo propone implicar directamente al estudiante en el proceso de desarrollo. A pesar de que la estrategia es fácilmente

automatizable, es importante implicar en el proceso de codificación a los estudiantes para que sean conscientes de la relación existente entre la gramática de atributos de partida y su subsecuente implementación, y mejorar así su aprendizaje y comprensión. El proceso de codificación propuesto se compone de dos etapas principales:

- En primer lugar, las reglas sintácticas de la gramática de atributos de partida se transcriben como reglas sintácticas de un esquema de traducción, procesable por una herramienta de generación de procesadores de lenguaje. En concreto, la estrategia propuesta propone la utilización de generadores de analizadores/traductores *bottom-up*, como CUP, aunque también es aplicable, con una mayor elaboración, a analizadores/ traductores *top-down*
- Posteriormente, se deben codificar las ecuaciones semánticas como acciones semánticas dentro del esquema de traducción. Por ello, la estrategia obliga a los estudiantes a transcribir las ecuaciones semánticas en términos de las dependencias entre atributos que se derivan de ellas, y del método de cálculo necesario.

Para que el proceso de cálculo de las implementaciones generadas, basadas en esquemas de traducción, sea fiel al proceso de cálculo de las gramáticas de atributo, se ha desarrollado una biblioteca software, llamada *EvLib*, que se empleará para especificar dichos cálculos como acciones semánticas en el esquema de traducción generado. De esta forma, los procesadores implementados preservarán los aspectos sintácticos y semánticos de la gramática de atributos de partida.

Por último, se han llevado a cabo diferentes experiencias con docentes y alumnos de un curso de Procesadores de Lenguaje para medir el grado de satisfacción entre éstos y evaluar la eficacia educativa de la estrategia. El enfoque de desarrollo obtuvo éxito entre docentes y estudiantes que lo encontraron útil a la hora de desarrollar sus propios procesadores de lenguaje. Por otro lado, la eficacia educativa observada pone de manifiesto la utilidad del enfoque frente a otras estrategias de desarrollo más convencionales y más frecuentemente usadas en un curso de Procesadores de Lenguaje.

5.1.3 Entorno educativo de desarrollo de procesadores de lenguaje basado en gramáticas de atributos

Como resultado integrador de este trabajo de tesis, se ha implementado también un entorno de desarrollo de procesadores de lenguaje basado en sus especificaciones en forma de gramáticas de atributos que satisface dos características principales:

- Facilitar la comprensión de las especificaciones mediante un depurador visual sofisticado. Dicho depurador muestra, de forma atractiva, el proceso de cálculo llevado a cabo por los procesadores sobre la gramática de atributos que especifica el procesador y una sentencia del lenguaje que dicha gramática define, dibujando el árbol sintáctico decorado y animando el proceso de cálculo sobre dicho árbol para que se visualice cómo se

mueven los valores semánticos por dicho árbol. Este enfoque está basado en los trabajos relacionados con *Evaluators*.

- Generar implementaciones como esquemas de traducción siguiendo los patrones de codificación propuestos en la estrategia de implementación contemplada en el punto anterior. De esta forma, los estudiantes serán capaces de identificar las implementaciones generadas con las especificaciones de partida, descritas mediante el formalismo de las gramáticas de atributos. Esta característica está basada en la estrategia para la implementación de gramáticas de atributos no circulares como esquemas de traducción y hace uso de la biblioteca *EvLib*.

Finalmente, se ha medido la utilidad percibida del entorno por docentes y estudiantes en un curso de Procesadores de Lenguaje. Los resultados muestran que ambos, docentes y estudiantes, encuentran el entorno de desarrollo útil y una herramienta complementaria muy adecuada para un curso de Procesadores de Lenguaje.

5.2 Trabajo futuro

Esta sección finaliza este trabajo de tesis exponiendo las líneas de trabajo futuro más prometedoras, derivadas de las propuestas presentadas en dicha tesis. Dichas líneas de trabajo futuro se enumeran a continuación:

- Desarrollo adicional de *Evaluators*.
- Explorar la aplicabilidad del modelo de proceso para el desarrollo de sistemas educativos en otros dominios.
- Experimentar los efectos en el aprendizaje de los estudiantes de la estrategia de desarrollo de procesadores de lenguaje a largo plazo.
- Desarrollo adicional de *EvDebugger.*
- Distribución en abierto de las herramientas y el software desarrollados.

Los siguientes puntos motivan cada una de estas líneas de trabajo futuro e investigación.

5.2.1 Desarrollo adicional de *Evaluators*

En esta línea de investigación se proponen diferentes mejoras relativas a las herramientas software que componen el sistema educativo *Evaluators*, que da soporte a la estrategia educativa orientada a mejorar el aprendizaje del formalismo de las gramáticas de atributos. Así mismo, se propone realizar estudios más extensivos sobre la estrategia educativa, las herramientas software del sistema y los simuladores generados con el objetivo de identificar posibles aspectos a mejorar en ellos.

5.2.1.1 Mejoras del sistema educativo *Evaluators*

Las mejoras relativas a la herramienta de autoría se resumen en mejorar el editor de gramáticas de atributos con capacidades similares a un entorno de desarrollo donde se proporcionan herramientas de autocorrección y auto-

completado. Además, al estar describiendo procesadores de lenguaje, también sería interesante presentar mensajes de depuración más sofisticados que ayuden a los docentes a identificar posibles errores en sus especificaciones.

Por otro lado, las mejoras relativas a la herramienta de análisis se centran en proporcionar funcionalidades para el procesamiento masivo y automático de soluciones. Este procesamiento detectará los fallos más comunes realizados en los ejercicios propuestos para que el docente pueda identificarlos y poder planificar acciones educativas para reforzar los conceptos implicados en dichos fallos.

Por último, ambos tipos de simuladores generados son también objeto de posibles mejoras. En concreto, una posible futura mejora que implica a ambos está relacionada con el *feedback* presentado por ambos. Algunos estudiantes, durante las experiencias realizadas, expusieron que la identificación de los errores cometidos durante la simulación eran, en ocasiones, difíciles de interpretar. Por otra parte, docentes de la asignatura de Procesadores de Lenguaje también apuntaron a dicha carencia de los simuladores generados (en concreto, docentes e investigadores de la Escuela de Informática de la universidad de Utrecht (Países Bajos), los cuales apuntaron a dicho problema durante una visita corta realzada por el autor de esta tesis al grupo de investigación del Prof. Johan Heuring, en el contexto del proyecto del Plan Nacional de I+D+i TIN2010-21288-C02-01, "Un Enfoque Generativo para el desarrollo de Herramientas de Producción y Despliegue de Objetos de Aprendizaje en el Campus Virtual"). Por ello, proponemos como trabajo futuro, incorporar a los simuladores generados un tutor experto basado en reglas que sea capaz de hacer sugerencias útiles para guiar la resolución de los ejercicios por cada fallo que el estudiante cometa durante la simulación. Dicha línea de investigación podrá realizarse, como parte de una colaboración futura, con los citados investigadores de la universidad de Utrecht.

5.2.1.2 Estudio comparativo entre simuladores generados con *Evaluators*

Esta línea de investigación surge como una necesidad para identificar las fortalezas y debilidades de los dos tipos de simuladores generados con *Evaluators*. En este trabajo de tesis se han mostrado resultados de evaluación de cada uno de los simuladores frente al método tradicional para resolver los ejercicios propuestos en la estrategia educativa formulada. De estas experiencias se ha observado que la satisfacción entre docentes y estudiantes frente a estos simuladores era buena, así como la eficacia educativa de ambos simuladores frente al método tradicional. Sin embargo, creemos que un estudio comparativo entre ambos tipos de simuladores nos proporcionará un punto de vista diferente de estudiantes y docentes, de donde podremos identificar fortalezas y carencias en ambos tipos de simuladores. De esta forma, seremos capaces de planificar futuras mejoras en ambos simuladores que los harán más efectivos y atractivos.

5.2.2 Aplicabilidad del modelo de proceso en otros dominios

La línea de trabajo futuro que se presenta aquí ha sido brevemente mencionada en este trabajo de tesis, ya que es uno de los motivos por los que se creó el modelo de proceso para el desarrollo de sistemas software educativos.

Como objetivo inmediato se espera poder aplicar el modelo de proceso para desarrollar un sistema software educativo que facilite la adquisición de conceptos básicos del formalismo de los esquemas de traducción. Sin embargo, dicho modelo de proceso puede ser aplicable a otros campos de la enseñanza dentro de la Ingeniería en Informática como algoritmia, estructuras de datos o lógica formal.

Incluso hemos planificado la aplicación del modelo de proceso en otros ámbitos distintos a la Informática, como la enseñanza de análisis literario, clasificación de corpus de documentos históricos o liderazgo en educación. La posibilidad de trabajar en otros ámbitos educativos podrá derivar en mejoras sustanciales en el modelo de proceso propuesto, al interactuar con docentes y estudiantes con una perspectiva y necesidades educativas diferentes a las encontradas en el ámbito de la Ingeniería en Informática. Esto, sin duda, enriquecerá el modelo de proceso y lo convertirá en un modelo más versátil. Estos desarrollos podrán llevarse a cabo dentro de los proyectos de investigación en el marco de las humanidades digitales y del e-learning que están actualmente activos en el grupo de investigación ILSA: el proyecto HUM14_251 “Modelo Unificado de Gestión de Colecciones Digitales con Estructuras Reconfigurables: Aplicación a la Creación de Bibliotecas Digitales Especializadas para Investigación y Docencia”, concedido por la Fundación BBVA en su Convocatoria 2014 de Ayudas a Proyectos de Investigación, y el proyecto TIN2014-52010-R “Repositorios Educativos Dinámicamente Reconfigurables en Humanidades”, concedido en la convocatoria 2014 del Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad. En el marco de dichos proyectos podremos planificar y ejecutar el desarrollo de software educativo relativo a los temas objetivo de los proyectos mediante la aplicación directa del enfoque de desarrollo de software educativo propuesto en este trabajo de tesis.

5.2.3 Experiencias a largo plazo de la estrategia de desarrollo de procesadores de lenguaje

En este trabajo de tesis, se han presentado diferentes resultados empíricos relativos a la estrategia para la implementación de gramáticas de atributos no circulares como esquemas de traducción. Dichos resultados fueron fruto de experiencias puntuales con estudiantes y docentes planificadas como actividades aisladas dentro de un curso de Procesadores de Lenguaje. A pesar de que los resultados de dichas experiencias fueron satisfactorios, es necesario observar el efecto educativo de dicha estrategia de desarrollo durante un curso completo de Procesadores de Lenguaje. Por ello, se plantea esta línea de trabajo futuro, como un estudio de valoración de la efectividad educativa de la estrategia de desarrollo formulada donde los estudiantes deberán desarrollar los procesadores de lenguaje pedidos durante el curso utilizando dicha estrategia. De esta forma, obtendremos suficiente evidencia empírica para valorar los efectos educativos a largo plazo de la estrategia de desarrollo dentro de un curso de Procesadores de Lenguaje.

5.2.4 Desarrollo adicional de *EvDebugger*

Esta línea de investigación se centra en dos aspectos principales: mejorar el entorno de desarrollo basado en gramáticas de atributos *EvDebugger* y realizar estudios de la eficacia educativa de dicha herramienta.

5.2.4.1 Mejora del entorno *EvDebugger*

La versión de *EvDebugger* presentada en este trabajo de tesis, es una versión temprana que muestra algunas deficiencias de uso y accesibilidad que deberán ser corregidas en futuras versiones. Una de las mejoras más relevantes, que se plantea como línea de trabajo futuro, es proporcionar mensajes de depuración de errores más sofisticados. Estos mensajes deberán proporcionar información útil durante la especificación de los procesadores de lenguaje, durante su compilación y durante su simulación a través del proceso de depuración visual. Por otro lado, es necesario mejorar el depurador visual con un método para seleccionar puntos de acceso rápido durante el proceso de depuración. De esta forma los usuarios podrán acceder a puntos concretos del proceso de cálculo de los atributos semánticos sin la necesidad de contemplar todo el proceso. Finalmente, se está estudiando ampliar el tipo de gramáticas procesables por *EvDebugger* e incluir la posibilidad de procesar gramáticas ambiguas.

Todas estas mejoras tienen su origen en la estancia breve realizada en la Universidade do Minho (Braga, Portugal), y el desarrollo de las mismas podrá ser llevado a cabo como futuras colaboraciones con el grupo de investigación de acogida durante la estancia breve: gEPL -- Grupo de Especificação e Processamento de Linguagens, dirigido por el Prof. Pedro Rangel Henriques.

5.2.4.2 Estudios de la eficacia educativa de *EvDebugger*

Finalmente, esta línea de investigación propone medir la eficacia educativa de *EvDebugger* como herramienta software de apoyo durante un curso de Procesadores de Lenguaje. En este trabajo de tesis, se han presentado resultados obtenidos de una experiencia de valoración llevada a cabo entre docentes y estudiantes de un curso de Procesadores de Lenguaje en la anteriormente citada Universidade do Minho. Estos resultados son prometedores, ya que se puede observar que *EvDebugger* ha cosechado éxito entre docentes y estudiantes de dicho curso. Por ello, esta línea de trabajo propone realizar diferentes estudios para medir la eficacia educativa de *EvDebugger*, tanto a corto como a largo plazo, para obtener una evidencia empírica del potencial educativo de esta herramienta. Además, de dichos estudios podremos obtener información relevante que podrá derivar en posibles mejoras en *EvDebugger* para convertir el entorno de desarrollo en una herramienta software educativa más efectiva y atractiva para los estudiantes.

5.2.4.3 Distribución en abierto del software desarrollado

Por último, como actividad de difusión y transferencia asociada a esta tesis, tenemos previsto la distribución en abierto de las distintas herramientas y software desarrollados en la misma (*Evaluators*, *EvLib*, *EvDebugger*), a fin de propugnar su uso por parte de la comunidad de educadores y alumnos en las materias relacionadas con el diseño y la implementación de los lenguajes informáticos

Capítulo 6: Artículos presentados

A continuación se incluyen los artículos editados que se aportan como parte de esta tesis doctoral.

6.1 Introducing a design-preserving implementation strategy in a compiler construction course

Cita completa:

Rodríguez-Cerezo D., Sierra, J.L. Introducing a Design-Preserving Implementation Strategy in a Compiler Construction Course. En SIIE'13: Actas del XV Simposio Internacional de Informática Educativa, 24-29. 2013.

Resumen original de la contribución:

Students of Compiler Construction courses often find it difficult to develop the practical projects that usually complement these courses. These projects typically consist of the development of a compiler for a reduced, but complete, programming language. In order to make our students aware of the importance of systematic engineering approaches, when addressing the projects they must clearly separate the specification and implementation aspects. Thus, first they must specify their compilers by using attribute grammars. Then they must develop their processors from these specifications by means of systematic transformations. During this process, students can make different mistakes that can hinder their progress. In order to address this problem, we have re-designed our pedagogical strategy to promote design-preserving implementations. For this purpose, we have developed a tiny library that, when used along with a parser generation tool, makes it possible to develop an initial executable language processor by directly encoding the specification. In this paper we describe this pedagogical strategy, as well as different assessment experiences conducted with instructors and students.

Referencia de citas bibliográficas:

[5][6][14][19][26][49][66][74][89][91][97][109][120][124][143][150][154][173]

Introducing a Design-Preserving Implementation Strategy in a Compiler Construction Course

Daniel Rodríguez-Cerezo
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases s/n
28040 Madrid (Spain)
+34913947506
drcerezo@fdi.ucm.es

José-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases s/n
28040 Madrid (Spain)
+34913947548
jlsierra@fdi.ucm.es

ABSTRACT
Students of Compiler Construction courses often find it difficult to develop the practical projects that usually complement these courses. These projects typically consist of the development of a compiler for a reduced, but complete, programming language. In order to make our students aware of the importance of systematic engineering approaches, when addressing the projects they must clearly separate the specification and implementation aspects. Thus, first they must specify their compilers by using attribute grammars. Then they must develop their processors from these specifications by means of systematic transformations. During this process, students can make different mistakes that can hinder their progress. In order to address this problem, we have re-designed our pedagogical strategy to promote design-preserving implementations. For this purpose, we have developed a tiny library that, when used along with a parser generation tool, makes it possible to develop an initial executable language processor by directly encoding the specification. In this paper we describe this pedagogical strategy, as well as different assessment experiences conducted with instructors and students.

Categories and Subject Descriptors
K.3 [**Computers and Education**]: (K.3.1) Computer Issues in Education – *Computer-assisted Instruction (CAI)*; (K.3.2) Computer and Information Science Education – *Computer science* education. D.3 [**Programming** Languages]: (D.3.4) Processors – *Translator writing systems and compiler* generators.

General Terms
Human Factors, Languages.

Keywords
Implementation Strategy, Education in Compiler Construction, Attribute Grammars, Parser Generators.

1. INTRODUCTION
Successful Compiler Construction courses rely on a balanced combination of theory and practice. Both aspects are equally important and complementary, because without the practical part of the course, students could not understand the different theoretical aspects taught in lectures. In fact, for these courses, instructors usually propose the implementation of a compiler for a programming language as a final project for students. Although the language proposed is usually a reduced and simplified version of other full-flagged programming languages (e.g., PASCAL or C), in our experiences teaching one of these courses at the Complutense University of Madrid (UCM), Spain, we have realized that the students find it difficult to complete the project, which is endemically considered to be a complex task [7]. Indeed, while we have observed that students are usually able to understand and master each technique, concept and tool presented in lectures separately, we have also realized that they have difficulties in figuring out how these elements can be applied together in order to develop a working language processor.

In order to help students overcome these difficulties we promote a top-down learning approach in our course. First, we adopt a software engineering perspective, by describing the global tasks that make up the development process model that students must adopt to build a compiler. Then, we use these tasks to contextualize each concept taught in lectures. Also, in this process model we clearly separate the specification aspects of a language processor (lexicon, syntax, static semantics and translation) from those related to implementation. Then, we instruct our students on how the implementation of the language processor can be obtained from the specification by performing different systematic transformation steps. However, these steps are usually laborious and potentially error-prone for novices. Thus students can make mistakes, which may delay the production of an initial working version of the processor. This fact can potentially discourage students, since they fail to get real implementations from their specifications.

In order to overcome the aforementioned shortcomings, we have designed a new approach oriented to minimizing the chance to make mistakes, and therefore the time to obtain a working implementation. This approach, which is described in this paper, proposes a method that promotes straightforward patterns for encoding the specification designed by the students, therefore accelerating the production of the first executable version of their processors.

The most widely-used approach to laboratory projects in Compiler Construction courses is to implement a reduced programming language (e.g., COOL [2] or MINIML [4]). Although our approach has been applied to programming languages, it is actually extensible to any other computer language. In addition, typical Compiler Construction courses introduce specialized language implementation tools (e.g., parser generators like JavaCC [10], CUP [3] or ANTLR [14]), as well as IDEs specialized in language implementation (e.g., ANTLRWorks [5], or XText [6]) to help students face the complexities of Compiler Construction. Our approach can guide how to make use of these tools for educational purposes. Finally, there are several tools (e.g., LISA [12] or Silver [18]) able to generate working implementations from attribute grammar specifications. Our approach is not intended to substitute these tools, but as a distinct and somewhat complementary approach, aiming to combine their strengths with the pragmatism of conventional parser generators.

The rest of the paper is organized as follows. Section 2 contextualizes the approach by describing the learning setting in which it is applied. Section 3 details the approach itself. Section 4 reports on the assessment of this approach from three different perspectives: teacher assessment, student assessment and educational efficacy. Finally, section 5 provides some conclusions and lines of future work.

2. CONCEPTUAL BACKGROUND

In our courses we thoroughly promote the use of two grammar-based formalisms for specifying and implementing language processors: *attribute grammars* and *translation schemata*.

Attribute grammars constitute a widely used formalism for specifying the syntax and semantics of programming languages [9][13]. Figure 1 shows an example of attribute grammar formalizing the evaluation of simple arithmetic expressions. From this example it is apparent how an attribute grammar is formed by the following components:

```
s → expr
   s.v = expr.v
   expr.m = initMem();
expr → expr+ term
   expr0.v = expr1.v + term.v
   expr1.m = expr0.m
   term.m = expr0.m
expr → term
   expr.v = term.v
   term.m = expr.m
term → term * fact
   term0.v = term1.v * fact.v
   term1.m = term0.m
   fact.m = term0.m
term → fact
   term.v = fact.v
   fact.m = term.m
fact → NUM
   fact.v = toInt(NUM.lex)
fact → ID
   fact.v = getVal(ID.lex, fact.m)
```

Figure 1 Attribute grammar specification of an arithmetic expression language

- *Syntactic symbols*, which represent syntactic constructions. There are two types: *terminal symbols* related to the simplest structures in the language (such as `NUM`, `+` or `*` in Figure 1), and *non-terminal symbols* associated to composite structures in the language (such as `expr`, `term` or `fact` in Figure 1).
- *Syntactic rules* (`expr → expr + term` in Figure 1), which determine the structure of non-terminal symbols.
- *Semantic attributes*, which are associated to the syntactic symbols. These attributes are further divided into two categories: *synthesized attributes* (`v` of `expr` in Figure 1), which keep the semantic meaning of the symbols, and *inherited attributes* (`m` of `expr` in Figure 1), which keep contextual information needed to compute such meanings.
- *Semantic equations* (`expr0.v = expr1.v + term.v` in Figure 1), which are associated to each syntactic rule. These equations determine how to compute the value of the synthesized attributes of the rule's left-hand side (LHS), and of the inherited attributes of the rule's right-hand side (RHS).

Translation schemata, in turn, are another extension of context-free grammar that augments syntactic rules with *semantic actions*: chunks of code in a host programming language to be executed during parsing [1]. These actions can consult and update *semantic values* that augment the parsing state. There are many types of translation schemata, but the following two are especially relevant:

- *Bottom-up translation schemata*. These schemata assign a semantic action to each syntactic rule, which will be executed when the rule is used in a *reduction* during a bottom-up recognition of the input sentence[1] [1]. Figure 2(a) shows an example concerning the specification in Figure 1. It uses a YACC-like notation to refer to the semantic values ($$ refers to the semantic value of the rule's LHS, and $i the semantic value of the *i*th symbol in the RHS). This kind of description is related with YACC [17], Bison [11] or CUP [3] tools.

(a)

```
global m = initMem();
s → expr
  {$$.v = $1.v;}
expr → expr+ term
  {$$.v = $1.v + $2.v}
expr → term
  {$$.v = $1.v}
term → term * fact
  {$$.v = $1.v * $2.v}
term → fact
  {$$.v = $1.v}
fact → NUM
  {$$.v = toInt($1.lex)}
fact → ID
  {$$.v =
     getVal($1.lex, m)}
     --0—
```

(b)

```
global m = initMem();
s(↑v) → expr(v)
expr(↑vE) →
    {local vT;}
    term(vT)
    rexpr(vT, vE)
rexpr(↓vRi,↑vR) →
    {local vT;}
    + term(vT)
      rexpr(vRi+vT,vR)
rexpr(↓vRi,↑vR) →
      {vR = vRi;}
term(↑vT) →
    {local vF;}
    fact(vF)
    rfact(vF, vT)
rterm(↓vRi,↑vR) →
    {local vF;}
    * fact(vT)
      rterm(vRi*vF,vR)
rterm(↓vRi,↑vR) →
      {vR = vRi;}
fact(↓vF) →
    {local n;}
    NUM(n)
    {vF = toInt(n);}
fact(↓vF) →
    {Local i;}
    ID(i)
    {vF = getVal(I,m);}
```

Figure 2(a) Description of a bottom-up translator implementing the specification in Figure 1 with a translation scheme, (b) a translation scheme for an equivalent top-down translator (↓ precedes input parameters, and ↑ precedes output ones)

- *Top-down translation schemata*. These schemata interleave semantic actions in the rule's RHS, which are executed as they are discovered during the top-down recognition of the input sentence[2] [1]. In addition, if *recursive descent parsing* [1] is used, semantic values can be represented by adding input and output parameters to the syntactic symbols (in the case of terminal symbols, the output parameters -the only ones allowed- will be provided by the scanner). Figure 2(b) shows one of these top-down translation schemata based on the specification of Figure 1. Schemata of this kind are used by JavaCC [10] or ANTLR [14] to generate parsers.

Finally, it is worthwhile to notice that, while attribute grammars *declare* the meaning of the symbols by using equations, translation schemata describe how this meaning must actually be computed during parsing. Thus, attribute grammars result in a higher-level formalism than translation schemata. In particular, in

[1] A bottom-up parser recognizes input sentences by substituting rule RHSs for the corresponding LHSs (these operations are known as *reduce* operations)

[2] A top-down parser recognizes sentences by rule *expansions* (rule LHSs are replaced with the corresponding RHSs) and terminal matching on the next symbol in the input.

an attribute grammar it is not necessary to worry about the order in which the semantic attributes must actually be computed (indeed, this order can be deduced from the dependencies among attributes established by the semantic equations). On the other hand, translation schemata strongly rely on the execution order of semantic actions, since this order is determined by the parsing method. In consequence, attribute grammars are more suitable for the specification stage, while translation schemata are more suited for implementation purposes.

3. DESIGN-PRESERVING IMPLEMENTATION APPROACH

We have observed how, among all the activities addressed by our students for the construction of language processors, it is in *grammar transformation* where they find most difficulties. Indeed, this activity is not trivial at all. For instance, if the implementation is based on top-down parsing, it will be necessary to transform the underlying context-free grammar to eliminate left-recursion and (maybe) left-factoring [1] (the translation scheme in Figure 2(b), which implements the specification in Figure 1, is an example). Even worse, in addition to syntactic rules, these transformations will also affect semantic equations, which can imply each semantic attribute. In consequence, the process is error-prone, difficult to assimilate by students, and difficult to apply as the size of the specification increases, as is the case with the non-trivial languages used in laboratory projects. Thus, while we consider grammar transformation a valuable activity from a pedagogical point of view, it is also necessary to evaluate the trade-offs introduced by its complexity to students, which can become a worrisome de-motivational factor.

In order to address these shortcomings, we have introduced a novel implementation technique focused on the direct encoding of the attribute grammar on a parser generation tool. While the resulting implementation may not be as efficient as the one obtained with grammar transformations, it can be obtained in a more straightforward way, thus avoiding the aforementioned de-motivational factor. In addition, the use of conventional parser generation tools lets students get glass-box implementations of attribute grammars. They can trace and modify the operation of the resulting implementations, and they can evolutionarily transform these implementations into final, more efficient implementations [15][16]. This glass-box character and the amenability of evolutionary development differentiate our approach from more straightforward ones, like that provided by attribute grammar-based tools like [12][18].

This section describes the re-designed implementation strategy. For this purpose, section 3.1 provides an overview of the approach. Then, the following sections go into each aspect of this approach.

3.1 Overview of the Approach

Figure 3 shows the different activities that the students must carry out to obtain a working translator from the attribute grammar-based specification in our approach, which is based on our previous work on the implementation of attribute grammars using conventional compiler construction tools [15]. First, according to this workflow, they must write an auxiliary *attribution specification*, which is based in the syntactic structure of the attribute grammar, and all semantic attributes are substituted by the rule *attribution*, an abstract structure describing the dependencies among rule attributes, as well as the semantic functions to be used in the calculation of these attributes. Then, they must obtain an *attribution scheme* from the attribution specification. This translation scheme describes the dependencies among attributes, as well as the semantic functions to be applied in their evaluation. Next step is to program an *attribution module* providing implementations of the attribution functions. The structure of these functions follows systematic patterns based directly on the semantic equations of the corresponding rules. Then, students must provide a suitable *implementation of the semantic functions*. Finally, students can automatically *generate* the working implementation by using a suitable parser generation tool.

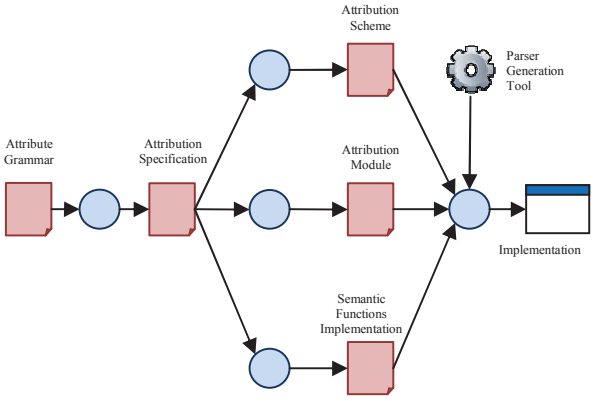


Figure 3 Diagram of the design-preserving implementation workflow

3.2 The *Evaluation* Library

The implementation workflow is facilitated by the use of a suitable *evaluation library*. This evaluation library is a set of services that let students (i) describe the dependencies among the attributes of the syntactic rules, (ii) install the semantic functions to be used to calculate the values of each attribute, and (iii) specify additional information that will be useful for debugging and tracing purposes. As a consequence of this description, dependencies between attribute instances will actually be created, enabling the subsequent computation of the values of these attributes.

In our course, we concreted this library in *EvLib*, a tiny Java library supporting a demand-driven semantic evaluation strategy [8], since our students use Java as the underlying implementation language. Figure 4 summarizes the structure of the library, which consists of six Java classes and interfaces:

- `SemFun` defines the interface to be followed by the semantic functions. Thus, semantic functions will be provided as Java classes that implement an `eval` method.
- `Attribute` is the base class for semantic attributes. It is further specialized in `LAttribute`, which represents attributes for terminal symbols, and `SAttribute`, which represents attributes for non-terminal symbols. Appart from the common `getValue` method for consulting its value, `SAttribute` provides methods to assign a semantic function (`setSFun`), and the dependencies from other attributes (`setDeps`).
- `SemTable` is a class used to contain the attributes assigned to a syntactic symbol. It includes a `putA` method for adding new attributes, and an `a` method for getting existing attributes.

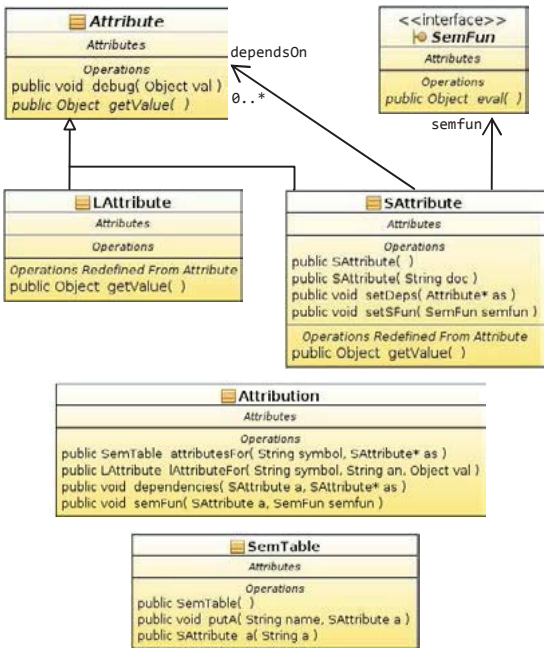


Figure 4 UML Class Diagram of *EvLib*

- `Attribution` is the base class for all the attribution modules (see subsection 3.4), which students must extend. This class actually implements a very simple internal domain-specific language to describe rule attributions specifying: syntactic rule (`rule`), LHS attributes (`attributesFor`), lexical attributes (`lAttributeFor`), semantic functions (`semFun`), and dependencies among attributes (`dependencies`).

The result is a very compact and simple library (less than 150 lines of source code), which we actually provide to our students in source form. They can use the services in the `Attribution` class to actually instrument the initial implementations in a very straightforward and quick way.

3.3 Writing the *Attribution Specification* and the *Attribution Scheme*

Armed with a suitable evaluation library (like *EvLib*), students can undertake implementation in a systematic way. For this purpose, the first activity to do is to write the attribution specification. This activity has the following objectives:

- To associate a semantic attribute *a* with each non-terminal symbol *A*, in order to contain attributions of rules $A \rightarrow \alpha$.
- To associate a semantic equation $A.a = \phi(\alpha.\mathbf{a})$ with each syntactic rule $A \rightarrow \alpha$, which represents the attribution of such a rule in the original specification. This attribution is in turn constructed from the attributions $\alpha.\mathbf{a}$ attached to the symbols in α.

Thus, this activity can be carried out in a straightforward way (indeed, since actual attribution will be coded within attribution functions like ϕ, this specification is equivalent to specifying how to synthesize abstract syntax trees). Figure 5 shows the attribution specification for the attribute grammar in Figure 1.

```
s → expr
   s.a = sA1(expr.a)
expr → expr+ term
   expr0.a = exprA1(expr1.a, term.a)
expr → term
   expr.a = exprA2(term.a)
term → term * fact
   term0.a = termA1(term.a, fact.a)
term → fact
   term.a = termA2(fact.a)
fact → NUM
   fact.a = factA1(NUM.lex)
fact → ID
   fact.a = factA2(ID.lex)
```

Figure 5 Attribution specification for the specification in Figure 1

```
s → expr
   {$$.a = sA1($1.a)}
expr → expr+ term
   {$$.a = exprA1($1.a, $3.a)}
expr → term
   {$$.a = exprA2($1.a)}
term → term * fact
   {$$.a = termA1($1.a, $3.a)}
term → fact
   {$$.a = termA2($1.a)}
fact → NUM
   {$$.a = factA1($1.lex)}
fact → ID
   {$$.a = factA2($1.lex)}
```

Figure 6 Attribution scheme derived from Figure 5

Once the attribution specification is available, the next step is to describe a translator that implements it. The result is the aforementioned *attribution scheme*. Being a translation scheme, its nature will be dependent on the particular parsing method adopted. In our course, we have promoted the use of bottom-up translators for this purpose, since they are able to accept virtually all the grammars that arise in laboratory projects. Thus, students can directly implement the attribution specifications without worrying about transformations (Figure 6 exemplifies this fact with the attribution scheme associated to Figure 5).

```
public class ExAttribution extends Attribution {
  private final static ADD = new AddSF();
  private final static IDEN = new IdenSF();
  private final static VALOF = new ValOfSF();
...
 public SemTable exprA1(SemTable expr1, SemTable term) {
   rule("expr ::= expr + term");
   SemTable expr0 = attributesFor("expr0","m", "v");
   dependencies(expr0.a("v"), expr1.a("v"), term.a("v"));
   dependencies(expr1.a("m"), expr0.a("m"));
   dependencies(term.a("m"), expr0.a("m"));
   semFun(expr0.a("v"),ADD);
   semFun(expr1.a("m"),IDEN);
   semFun(term.a("m"),IDEN);}
...
 public SemTable factA2(String lexNum) {
   rule("fact ::= NUM");
   SemTable fact = attributesFor("fact","m", "v");
   LAttributelexOfNum =
         lAttributeFor("NUM","lex",lexNum);
   dependencies(fact.a("v"), fact.a("m"), lexOfNum);
   semFun(fact.a("v"),VALOF);}
```

Figure 7 Excerpt of the attribution module for Figure 1

3.4 Writing the *Attribution Module*

Once the attribution scheme is available, students must provide an *attribution module* with the actual implementation of attribution functions. In order to write each attribution function, they must use the services of the attribution library to: (i) declare the rule which is being attributed (this information will be used solely for documentation and tracing purposes), (ii) declare the attributes for the rule LHS, (iii) create the lexical attributes for the terminal symbols, (iv) describe the dependencies among attributes (these descriptions will be derived from semantic equations in a

straightforward way), and (v) establish the semantic function to be used in the computation of each attribute.

In our case, and as indicated earlier, our students must provide these semantic modules as Java classes by extending `Attribution` in the *EvLib* library. Figure 7 shows an excerpt of the attribution module for the specification in Figure 1. In particular, it shows the implementation of `exprA1` and `factA2` attribution functions.

3.5 Writing the Semantic Functions

In order to finish the implementation, students must implement the semantic functions referred by the original specification. In our case, these semantic functions must implement the `SemFun` interface in *EvLib*, and, indeed, the `eval` method. This method takes the sequence of attributes involved in the computation as input. Then: (i) It must recover the values of the attributes, and (ii) it must carry out the computation.

```
class AddSF implements SemFun {
   public eval(Attribute a0, Attribute a1) {
     int v0 = (Integer)a0.getValue();
     int v1 = (Integer)a1.getValue();
     return v0+v1;
   } }
class IdenSF implements SemFun {
   public eval(Attribute a) {
     return a.getValue();
   } }
class ValOfSF implements SemFun {
   public eval(Attribute am, Attribute aid) {
     String id = (String)aid.getValue();
     Map<String,Integer> m = am.getValue();
     return m.get(id);
   } }
```

Figure 8 Examples of semantic functions

Figure 8 shows the implementation of the semantic functions referred to in Figure 1.

3.6 Generating the Translator

Once all the required components are available, students can generate the translator by using a suitable parser generation tool. In particular, our students used CUP (along with JFlex for writing the scanner) [3].

4. ASSESSMENT OF THE APPROACH

In order to assess our design-preserving implementation approach, we conducted three different evaluation efforts. First, we conducted an informal survey directed at instructors familiar with the Compiler Construction Course, who assessed the strengths and weaknesses of the approach. Then, we organized a survey to students, who are already familiar with a more conventional implementation strategy, in our introductory Compiler Construction at UCM. Finally, we performed a controlled experiment involving two groups of volunteer students from the aforementioned Compiler Construction course, in order to assess the educational efficacy of the approach. Next sections summarize the results of these evaluations.

4.1 Evaluation by Instructors

When we finished the design of the educational approach, we informally interviewed six Computer Science instructors familiar with the topic, explaining to them the main ideas behind our new implementation proposal, showing them some implementation examples, and asking them for their opinions.

Most of the instructors highlighted, as excellent features, the clarity of the code developed by using the library and the way in which the implementation patterns mirror the original specifications. Some instructors also indicated how the systematic patterns applied could contribute to evaluating laboratory work and to better identifying potential mistakes in the implementations. On the other hand, some instructors interviewed asked us why we did not use tools like LISA [12] or Silver [18], which directly support attribute grammars. However, once we highlighted the glass-box and evolutionary development aspects enabled by our approach, they recognized the different aim of our proposal. Besides, most of the instructors, after recognizing the relative sophistication of the approach, thought the students could get into trouble once something went wrong (e.g., by forgetting to specify some dependency).

4.2 Evaluation by Students

In order to assess our approach with students, during the current 2012-2013 academic year we conducted an experience consisting of the implementation of a specification concerning code generation for the short-circuit evaluation of Boolean expressions [1]. We recruited volunteers among our students in the Compiler Construction course (participation was rewarded as extra credit). Then we collected the opinion of the students working with the novel approach by means of a short survey containing four questions with answers based on a four-point Likert-like scale (*Disagree, Partially Disagree, Partially Agree, Agree*). The survey items are listed in the first column of Table 1; the second column registers the percentage of students that chose *Agree* and *Partly Agree* as an answer.

Table 1 Results obtained from the opinion test

Survey Items	***% Agreement***
I found it easy to use the tools provided to develop the language processor	83.16
I found it easy to debug the language processor	52.29
Compared to the techniques used to develop the project in the 1st semester, I find it easier to develop using the tools provided	90.77
I'm considering using the tools provided to develop my next language processors	77.75

More than 80% of students surveyed found the new development method easy and intuitive, and more than 90% found it easier than the techniques explained during the first semester (top-down one-pass translation, and heavy use of grammar transformations). The students' perceived difference between the two development methods is such that more than 77% were considering using it for the second part of the project, to be finished during the second semester. On the negative side, it is noticeable how a significant number of students found the implementations difficult to debug (in consonance with instructors' perception). Thus, it is a clear aspect to improve in the near future.

4.3 Educational Efficacy

In order to assess the educational efficacy of the approach, we split volunteers in two groups on the basis of the grades obtained so far in the course. A first group of 21 students used JavaCC to deliver the final implementation. The other 29 volunteers (actually, those interviewed as described in the previous section) worked with the novel implementation technique. They were provided with a non l-attributed specification of the short-circuit code generation, based on the explicit propagation of jump addresses as inherited attributes [1]. Final implementations were generated using JFlex + CUP.

Once the work done by the students in the two groups was examined, we graded it while taking into account the clarity of the code, the application of systematic development principles, and the correctness of the grammar transformations, when needed. The students' performance was rated in three different categories: *excellent* (*Ex*), *satisfactory* (*Sat*) and *unsatisfactory* (*UnSat*).

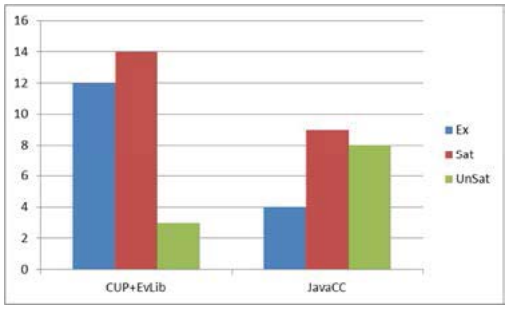


Figure 9. Barchart of students' performance

Figure 9 summarizes the results obtained. The chart shows the number of students, per development method, that obtained each rate. Thus, it can be observed that the design-preserving approach obtained better results than the more conventional, one-pass top-down translation, strategy (supported, in this case, with JavaCC). The majority of students who performed the experiment using the novel approach, CUP and *EvLib*, reached an excellent (41,37%) or satisfactory (48,27%) result. However, the majority of projects carried out using conventional top-down translation and JavaCC were rated as satisfactory (42,85%) or unsatisfactory (38,09%). In addition, the perceived improvement was statistically significant (χ^2 test: p-value = 0.044).

5. CONCLUSIONS AND FUTURE WORK

In this paper we have described a design-preserving implementation approach for laboratory projects in Compiler Construction courses. On one hand, the approach is systematically driven by attribute grammar-based specifications. On the other hand, it gives our students the opportunity to practice with conventional compiler construction tools (parser generators, in particular). Indeed, using our approach students can quickly and easily obtain an initial implementation from their specification that can be further refined to obtain more efficient processors. The approach was well received by instructors and students, although they also raised some aspects (e.g., better debugging facilities) that could be improved in future releases. Also, the educational efficacy of the approach was favorably tested with a controlled experiment that yielded significant evidence of how this approach improves a more conventional one, based on implementations of one-pass top-down translators.

We plan to further evaluate the approach in our Compiler Construction course at UCM. In particular, we plan to systematically assess its long-term benefits. Besides, we are working on improving our *EvLib* library in order to include better support for debugging, as suggested by several instructors and students during the assessment experiences.

6. ACKNOWLEDGMENTS

This research was partially supported by grants TIN2010-21288-C02-01 and EDU/3445/2011.

7. REFERENCES

[1] Aho, A.V., Lam, M.S., Sethi, R.&Ullman, J.D. 2007.*Compilers: principles, techniques and tools*(2nd edition).Addison-Wesley.

[2] Aiken, A. 1996. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7),19-24.

[3] Appel, A.W. 2002. *Modern Compiler Implementation in Java*. Cambridge University Press.

[4] Baldwin, D. 2003. A compiler for teaching about compilers.*34thSIGCSE technical symposium on Computer science education*(SIGCSE'03), 220-223.

[5] Bovet, J. and Parr, T. 2008.*ANTLRWorks: an ANTLR grammar development environment*. Software: Practice & Experience, 38, 1305–1332.

[6] Eysholdt, M.& Behrens, H. 2010. Xtext: implement your language faster than the quick and dirty way. ACM international conference on Object Oriented Programming Systems Languages and Applications Companion (SPLASH '10), 307-309.

[7] Griswold, W.G. 2002. Teaching software engineering in a compiler project course. *ACM Journal of Educational Resources in Computing*, 2(4).

[8] Jalili, F. 1983. A general linear-time evaluator for attribute grammars. *ACM SIGPLAN Notices*, 18(9), 35-44.

[9] Knuth, D.E. 1968. Semantics of Context-free Languages. *Mathematical System Theory* 2(2), 127-145.

[10] Kodaganallur, V. 2004. Incorporating language processing into Java applications: a JavaCC tutorial. *IEEE Software*, 21(4), 70-77.

[11] Levine, J. 2009.*Flex & Bison Text Processing Tools*. O'Reilly Media.

[12] Mernik, M.&Zumer, V. 2003.An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46, 61-68

[13] Paakki, J. 1995. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computer Surveys*, 27(2), 196-255.

[14] Parr, T. 2007.*The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf.

[15] Rodriguez-Cerezo, D., Sarasa-Cabezuelo, A. & Sierra, J-L. 2012. A systematic approach to the implementation of attribute grammars with conventional compiler construction tools. *Computer Science and Inf. Syst.* 9(3), 983-1017.

[16] Sarasa-Cabezuelo, A& Sierra, J-L. 2013.The grammatical approach: A syntax-directed declarative specification method for XML processing tasks. *Computer Standards &Interfaces* 35(1), 114-131.

[17] Schreiner, A.T. & Friedman, H.G. 1985. Introduction to Compiler Construction with Unix. Prentice-Hall.

[18] Wyk, E.V., Bodin, D., Gao, J., &Krishnan, L. 2010. *Silver: An Extensible Attribute Grammar System*. Science of Computer Programming, 75(1-2), 39-54.

6.2 User-centered development of generative educational systems for computer engineering: The *Evaluators* case study

Cita completa:

Rodríguez-Cerezo D., Sarasa-Cabezuelo A., Gómez-Albarrán M., Sierra-Rodríguez J.L. User-Centered Development of Generative Educational Systems for Computer Engineering: The Evaluators Case Study. International Journal of Engineering Education, 31(3): 751–763. 2015.

Resumen original de la contribución:

This paper describes the development of Evaluators, an educational system oriented to the generation of different kinds of interactive simulations for introductory Compiler Construction courses in Computer Science and Computer Engineering degrees. Evaluators consists of three main instructor-oriented components: (i) an authoring tool that instructors use to author collections of exercises concerning attribute grammars, which are a basic formalism for describing language processing tasks, (ii) a generator, which is able to automatically generate different kinds of interactive simulations from these batteries of exercises, and (iii) an analytic tool used to assess student performance by analyzing the logs recorded while students use the simulations. The underlying development principles behind Evaluators promote: (i) a generative approach to the production of interactive simulations from high-level descriptions of exercises, and (ii) a user-centered approach, according to which the system was iteratively enhanced according to the continuous assessment of the successive versions of the system with instructors and students. In this way, these principles illustrate a development approach centered on the construction of generative educational software (instead of on the construction of individual applications) and guided by a continuous assessment of this software with end users (not only students, but also instructors), which can be meaningfully extrapolated to other fields of Engineering Education.

Referencia de citas bibliográficas:

[5][3][9][17][27][28][35][38][41][42][43][51][55][59][61][67][87][92][93][104] [105][112][113][115][120][123][138][137][139][141][142][146][147][151][158] [163][162][170]

International Journal of Engineering Education Vol. 31, No. 3, pp. 751–763, 2015
Printed in Great Britain

0949-149X/91 $3.00+0.00

User-Centered Development of Generative Educational Systems for Computer Engineering: The *Evaluators* Case Study*

DANIEL RODRÍGUEZ-CEREZO, ANTONIO SARASA-CABEZUELO, MERCEDES GÓMEZ-ALBARRÁN and JOSÉ-LUIS SIERRA
Fac. Informática, Universidad Complutense de Madrid, C/ Prof. José García Santesmases 9, 28040 Madrid, Spain.
E-mail: drcerezo@fdi.ucm.es, asarasa@fdi.ucm.es, albarran@sip.ucm.es, jlsierra@fdi.ucm.es

This paper describes the development of *Evaluators*, an educational system oriented to the generation of different kinds of interactive simulations for introductory Compiler Construction courses in Computer Science and Computer Engineering degrees. *Evaluators* consists of three main instructor-oriented components: (i) an authoring tool that instructors use to author collections of exercises concerning attribute grammars, which are a basic formalism for describing language processing tasks, (ii) a generator, which is able to automatically generate different kinds of interactive simulations from these batteries of exercises, and (iii) an analytic tool used to assess student performance by analyzing the logs recorded while students use the simulations. The underlying development principles behind *Evaluators* promote: (i) a generative approach to the production of interactive simulations from high-level descriptions of exercises, and (ii) a user-centered approach, according to which the system was iteratively enhanced according to the continuous assessment of the successive versions of the system with instructors and students. In this way, these principles illustrate a development approach centered on the construction of generative educational software (instead of on the construction of individual applications) and guided by a continuous assessment of this software with end users (not only students, but also instructors), which can be meaningfully extrapolated to other fields of Engineering Education.

Keywords: development process model; application generators; computer science and computer engineering education; attribute grammars; compiler construction courses

1. Introduction

Interactive simulations are frequently used in engineering educational settings [1]. Indeed, they are powerful tools for learning and teaching procedures and processes in engineering. Additionally, they do not only focus on the contents, but also on the underlying mental processes required to understand basic concepts in engineering. They present these mental processes in an attractive and appealing way, which increases student motivation and engagement in the learning process [2]. The production of a suitable interactive simulator to teach any subject in engineering is a non-trivial task that requires the collaboration of specialists in different disciplines (instructors, graphic designers, software developers, etc.). We have indeed faced these problems with the development of *Evaluators*, a system oriented to teaching basic concepts in introductory Compiler Construction courses in Computer Science and Computer Engineering [3–6]. The main aim of *Evaluators* is to help students in learning the basic principles of *attribute grammars*, which are a high-level declarative formalism widely used in compiler construction courses to specify syntax-directed translation tasks [7]. For this purpose, *Evaluators* generates serious games and interactive simulations that let students actively participate in the computation of the semantic attributes of nodes in syntax trees occurring during the semantic evaluation processes involved in syntax-directed translation [7].

This paper is focused on the development of *Evaluators*. This development is based, on one hand, on a *generative approach* centered on the construction of *application generators* instead of on particular applications. On another hand, it is based on user-centered continuous assessment oriented to the iterative enhancement of the different components integrated in the resulting educational system. In our opinion, the underlying development process model (see [8] for a more formal presentation of this process model) may be useful for the development of similar systems in other engineering educational scenarios, in which meaningful learning goals can be accomplished by solving significant batteries of exercises, whose solutions can be meaningfully formulated in the form of interactive simulations.

The rest of the paper is organized as follows: section 2 outlines some works related to ours; section 3 gives a brief introduction to *Evaluators* as well as to its pedagogical underpinnings; section 4 outlines the development process model followed

* Accepted 24 October 2014.

during the development of *Evaluators*; section 5 describes the development of *Evaluators* itself; and finally, section 6 provides final conclusions and some lines of future work.

2. Related work

The complexity of developing educational software requires the use of software engineering techniques [9]. In order to undertake this, it is possible to find several approaches in the literature, some of which are summarized below:

- *Use of existing software engineering approaches for the development of educational software*. Some examples of these approaches, which promote the use of conventional elements from Software Engineering in the development of an e-Learning product, are described in [10, 11].
- *Instructional design approaches*. These approaches are based on well-established instructional design methods in order to yield methodologies specifically tailored to the development of educational applications. Perhaps one of the most well-known is ADDIE (Analysis-Design-Development-Implementation-Evaluation) [12], a methodology for the design of (not only computer-mediated, but also conventional) instructional systems. Other approaches are specifically tailored to the development of educational software (e.g., [13–15]).
- *Agile approaches*. Agile methods promote more flexible and adaptive development approaches. Some examples of works promoting the agile development of eLearning systems are described in [16–18].
- *Model and reuse-oriented approaches*. These approaches are based on explicit formulations of generic models for educational software, as well as on the explicit identification of reusable assets in order to facilitate the development of new applications. The works described in [19, 20] exemplify these approaches.

In contrast to these approaches described in the literature, ours suggests a development process mainly driven by the assessments that the system users (instructors and students) make. Besides, the educational software system developed using our methodology is based on a generative approach, which results in the direct involvement of the instructors in the development of the system by providing the exercise descriptions.

In addition, while these efforts described in the literature are oriented to developing any type of educational system and software, our proposal is more focused on the development of a particular kind of educational system: interactive educational simulations and game-like interactive simulations. For this purpose, our proposal can be related to other efforts regarding the development of game-based educational software and game-like educational interactive simulations. Some of these efforts include methodologies specifically tailored to the design of serious games [21–23], while others introduce reusable assets (e.g., models, frameworks, toolkits) that facilitate educational game development [24–27].

Our approach is close to these works on models, frameworks and toolkits for developing educational games, in the sense that they promote the generation of final games from high-level descriptions. However we adopt a more specific and narrow approach according to which instructors are released from having any expertise in game design and are focused on what they are expert in: devising quality learning materials. In this sense, our proposal is closer to [24], which proposes the use of a flexible framework that allows the customization of games using educational assets like exercises provided by instructors, although we do not comply with any underlying framework. In addition, we emphasize the integration of the assessment aspect throughout the different phases of the process.

Finally, our proposal can be meaningfully related to other approaches concerning the application of generative principles to the development of educational software:

- *Generative Learning Objects*. A Generative Learning Object is a pattern that represents a family of learning objects that share common features with the exception of the values of certain parameters (*adaptation values* or *contextualization*) [28]. Some representative examples of works concerning generative learning objects are described in [29–31].
- *Domain-specific languages in the educational domain*. Domain-specific languages let users develop their software systems through textual (or even visual) notations tailored to a particular domain. Some approaches adhering to the principles of domain-specific languages are [25, 26, 32, 33].

In this way, our approach is close to the marriage of generative learning objects and domain-specific languages in the educational domain. Indeed, educational games and interactive simulations could well be conceived as instances of the same generative learning object. For this purpose, parameterization of learning objects would be carried out using sophisticated domain models, incarnated in domain-specific languages, instead of plain collections of attribute-value pairs. Other works in the

field of compiler construction education adopting a similar approach are [34–36].

3. *Evaluators*

As we indicated earlier, the aim of *Evaluators* is the generation of interactive simulations oriented to enhancing the learning and teaching of fundamental concepts in introductory Compiler Construction courses. In particular, the system is focused on enhancing the teaching and learning of basic notions of *syntax-directed translation*. For this purpose, the system adopts the *attribute grammars* formalism [7] as the main paradigm for undertaking syntax-directed translation.

Figure 1 shows a simple example of attribute grammar concerning a simple language of numeric expressions that will support the addition and multiplication of integers, as well as the evaluation of these expressions (for this purpose, the use of a predefined constant table will be used). From this example it is apparent how an attribute grammar is made up of the following components:

- *Syntactic symbols*, which represent syntactic constructions. There are two types: *terminal symbols* related to the simplest structures in the language (such as `num`, + or * in Fig. 1), and *non-terminal symbols* associated with composite structures in the language (such as `exp`, `term` or `fact` in Fig. 1).
- *Syntactic rules* (e.g., `term` → `term` "*" `fact` in Fig. 1), which determine the structure of non-terminal symbols (in this case, the rule states that a `term` may be a simpler `term`, followed by the terminal symbol "*", followed by a `fact`).
- *Semantic attributes*, which are associated with the syntactic symbols. These attributes are further divided into two categories: *synthesized attributes* (e.g., the attribute `val` of `exp` in Fig. 1), which keep the semantic meaning of the symbols (in this case, the *value* that results from evaluating the expression), and *inherited attributes* (e.g., the attribute `ienv` of `exp` in Fig. 1), which keep contextual information needed to compute such meanings (in this case, the table that maps constants into their numeric values).
- *Semantic equations* (e.g., `exp(0).val = exp(1).val + term.val` in Fig. 1), which are associated with each syntactic rule. These equations determine how to compute the value of the synthesized attributes of the rule's left-hand side, and of the inherited attributes of the rule's right-hand side (in this case, the equation states that the value of the expression formed by the sum of a term and another, simpler expression is the result of adding the values of such a term and the simpler expression).

While attribute grammars constitute a high-level declarative formalism for the specification of language processing problems, from our experience teaching language processing topics at the Complutense University of Madrid we have realized that students encounter difficulties in understanding this type of formalism, especially those not accustomed to dealing with formal and declarative specification methods [6]. One of the main causes of this difficulty is the non-standard computational model underlying attribute grammars. Indeed, the computation of attribute values in nodes in parse trees is guided solely by the dependencies among attributes introduced by semantic equations. In this way, the order in which attributes are computed does not matter, provided that the dependency constraints are satisfied (i.e., an attribute is evaluated only when all the attributes on which it depends have been in turn evaluated). Thus, our main aim with *Evaluators* was to help students overcome these difficulties by assimilating this non-standard computational model in the early stages of the learning process. For this purpose, *Evaluators* lets instructors turn

```
s → exp
  exp.ienv = initTable()
  s.val = exp.val
exp → exp "+" term
  exp(0).val = exp(1).val + term.val
  exp(1). ienv = exp(0). ienv
  term. ienv = exp(0). ienv
exp → term
  exp.val = term.val
  term. ienv = exp. ienv
term → term "*" fact
  term(0).val = term(1).val * fact.val
  term(1). ienv = term(0). ienv
  fact. ienv = term(0). ienv
term → fact
  term.val = fact.val
  fact. ienv = term. ienv
fact → num
  fact.val = str2Int(num.lex)
fact → iden
  fact.val = getValue(fact.ienv,iden.lex)
fact → "(" exp ")"
  fact.val = exp.val
  exp. ienv = fact. ienv
```

Fig. 1. An example of attribute grammar.

batteries of exercises concerning attribute grammars into interactive simulations. Using these simulations, students are able to find suitable orders in which semantic attributes can be evaluated in syntax trees for valid sentences in the languages. In its current version, *Evaluators* makes it possible to generate two different types of interactive simulations (a game-like one, and a more straightforward, visualization-oriented one) from the same collection of exercises. In addition, *Evaluators* also provides an instructor-friendly authoring tool, which can be used by instructors to author collection of exercises, as well as an analytic tool, which can be used by these instructors to analyze student achievements. Next sections give more details on the system and on its development (see also [3–6] for more in-depth descriptions of *Evaluators*).

4. The development approach of *Evaluators*

The development approach followed in the construction of *Evaluators* encourages the development of generative educational systems able to automatically generate educational simulations from representative collections of exercises created by instructors (see [8] for a more formal treatment of the approach). Instructors design the educational content, in the form of collections of exercises, while the actual simulations are automatically generated from these collections of exercises. This approach grants a certain level of customization of the educational artifacts generated without the assistance of a development team. Figure 2 shows a workflow that summarizes the development approach followed to create *Evaluators*. Next, we dissect the different phases that make up the development approach as well as the products obtained (see [8]):

- The production process starts with the *Conception* phase, where instructors and developers decide what kind of exercises will be contemplated in the system and what type of interactive simulations will be generated by the system.
- The following phase is the *Tool provision*, where developers create the toolset that will make up the educational system. The tools are the *Authoring tool*, the *Simulator generator* and the *Analytic tool,* which will play an essential role in the generation and exploitation of the educational simulations.
- Once the tools are available, in the *Exercise authoring* phase instructors create a collection of *exercises* using the authoring tool, which is a specialized editor for the kind of exercises that will be contemplated in the system.
- Then, in the *Simulation generation* phase instructors generate the educational simulations from the exercises that they created in the *Exercise authoring* phase. To generate the simulations, the instructors use the *Simulator generator.*
- Interactive simulations will be used by students in order to solve the exercises during the *Exercise resolution* phase. In addition, the approach supports the production of simulations that record student behavior in significant *logs*.
- Finally, instructors can exploit the *logs* produced during the exercise resolution in order to assess student performance. It is done in the *Student assessment* phase using the *analytic tool*. By exploiting the logs produced by the interactive simulation in the *Exercise resolution* phase, the analytic tool helps instructors study the students' behavior after using the simulation in order to assess students' progress and identify the leaks and problems they found. The results of this analysis are registered in the *grades* report, in which instructors adequately grade students.

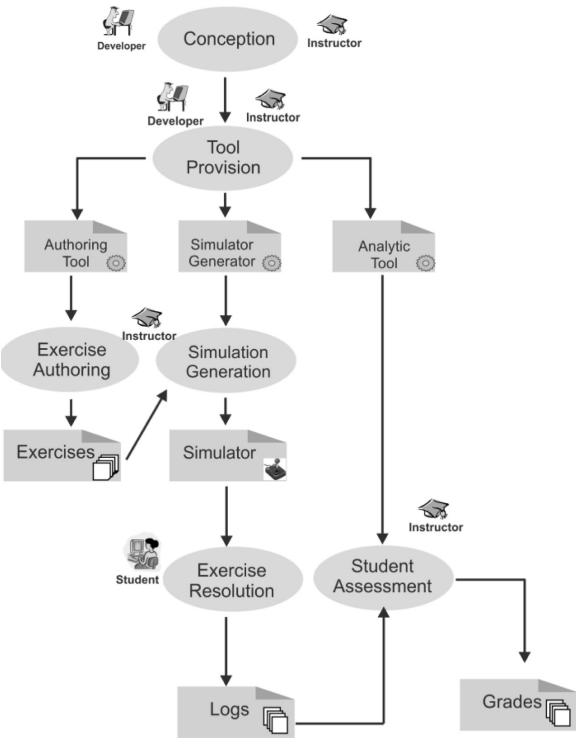


Fig. 2. Workflow summarizing the development approach of *Evaluators*.

In addition, the approach contains a continuous assessment process implicitly associated to the main processes described above. This assessment process is oriented to improving the different aspects of the resulting educational systems [37]. This way, instructors and students, through their opinions and assessments, can cause the reactivation of the development phases in the production process to improve the products that compose the system,

adjusting them to suit their needs. Each assessment activity is guided by two main assets: *Assessment instruments*, which capture all the information and procedures required to carry out the assessment process, and *Assessment reports*, which record the results obtained from the application of the assessment instruments and are eventually used to restart the corresponding phases in order to improve their resulting products. The assessment instruments and reports of a given product are designed when the product is built. The assessment instrument is used in those phases where the associated product is employed, in order to collect the users' opinion (instructors or students). The results of applying an assessment instrument are registered in the corresponding assessment report, which will be used in the phase where the product was created to redesign or adjust it. Finally, we should again point out that assessment instruments could be also the object of enhancement any time in the production process.

5. Detailing the development of *Evaluators*

The following sections describe in depth the development of *Evaluators*, from its initial version to its current version.

5.1 Conception phase

We began the development of *Evaluators* by setting the kind of exercises and interactive simulations pertinent to the system with the help of instructors in the subject, during the 2009–2010 academic year. In this first version, the exercises on syntax-directed translation tasks proposed to the students were defined by:

- Both an informal and a formal (attribute grammar) description of a language processing task.
- A sentence in the language to be processed.
- The attributed syntax tree built by processing the sentence according to the attribute grammar provided.

As a solution to an exercise, students had to provide a valid evaluation order of the attribute instances that decorated the corresponding syntax tree.

Likewise, in collaboration with instructors, we decided to set the interactive simulations generated as serious games where the students were situated in a maze-like virtual world based on the structure of the syntax tree. Inside this maze the attributes were represented as boxes that students had to move through the maze, in order to emulate the evaluation process (see [3, 6] for more details).

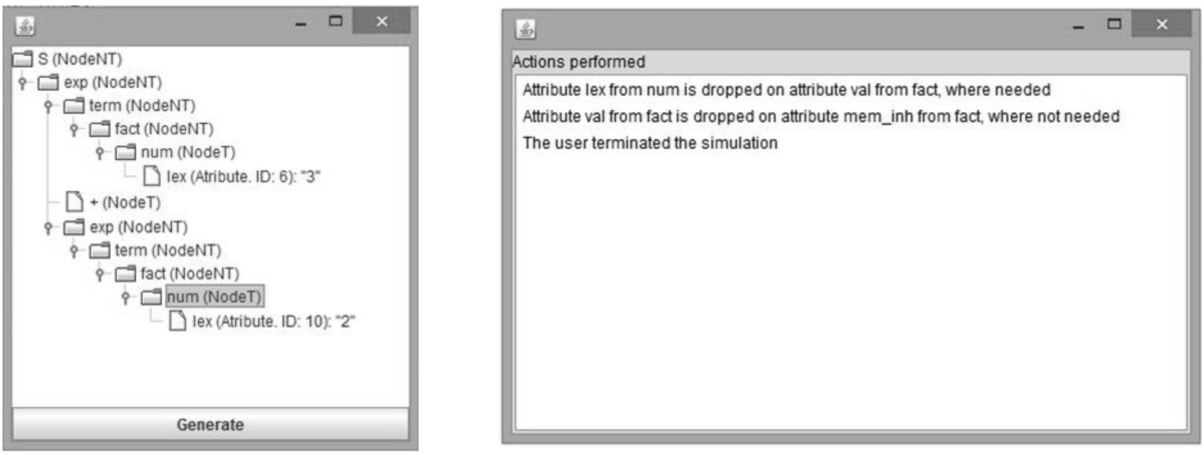
The assessment instruments necessary to evaluate the quality, adequacy and educational effectiveness of the educational games and the type of exercises were also defined with the help of instructors. The aspects to evaluate in the interactive game-like simulators were the mechanics of the game, the sense of progress, use, utility and interface of the different features provided, and the adaptation to the exercise type. The aspects evaluated of the exercise were the adequacy to the students of the information supplied, the solutions sought, and the educational utility of the exercise type. Since in this first iteration the evaluations would be carried out by a small group of instructors, informal meetings were the setting selected to evaluate the aspects of both exercises and educational games.

5.2 Tool provision phase

During the *Tool provision* phase, we developed the three software tools that would make up the system: the *authoring tool*, *simulator generator* and *analytic tool* (for a detailed description of each tool see [3]):

- The first version of the *authoring tool* consisted of a visual editor where the instructor could specify all the data required for the exercises proposed in the previous phase: the statement of the exercises (informal and formal description of the syntax-directed processing task), the sentence to be processed and the corresponding attributed syntax tree [8]. The instructor had to create the syntax tree, the attribute instances for each node and the dependencies among them, in order to represent all the constraints to be satisfied by the feasible solutions [7, 38]. This way, instructors were compelled to create the tree nodes and interconnect them. Fig. 3a shows a snapshot of this authoring tool modeling a sentence in the language addressed in Fig. 1 (specifically, the sentence is "3+2").
- Likewise, the *analytic tool* showed a plain text description of the actions carried out by students (see Fig. 3b).
- Finally, the *simulator generator* was responsible for processing the exercises created with the authoring tool to assemble an interactive game-like simulator according to the game description set out above.

Assessment instruments for each tool were also created. Some aspects contemplated in the three assessment instruments were the usability of the corresponding tool and the adequacy of the tool to its task. More specifically, the assessment instrument of the analytic tool contemplated aspects like the adequacy to the type of solutions and the quality of the information presented. We included aspects such as the expressivity provided to define the exercises and the appropriateness to the type of exercises in the assessment instrument of the authoring tool. Finally, the assessment instrument of the



(a) (b)

Fig. 3. (a) Snapshot of the first *authoring tool*; (b) Snapshot of the *analytic tool*.

simulator generator also contained aspects such as the adequacy to the educational artifact type and the quality of the mazes generated. The procedure selected was, once more, informal meetings in all the cases.

5.3 Exercise authoring phase

The *Exercise authoring* phase was first carried out during the summer of 2010 with a small group of instructors. They had to encode a battery of exercises from an existing set of paper-based exercises using the authoring tool created in the previous phase. We had previously instructed them in the operation of the tool.

We conducted informal meetings with the instructors in order to assess the authoring tool. As a positive point, the instructors reported that the tree-view of the authoring tool gave them a clear idea of the exercise's complexity. Moreover, the instructors evaluated the kind of exercises proposed. Instructors noted that the exercise model was consistent with the content taught in the first lessons about attribute grammars and it had all the information needed by the students to obtain the solution on their own. However, the assessment reports showed that it was very easy to make mistakes when specifying an exercise with the tool. Indeed, instructors reported that the tool lacked practical uses when creating complex exercises (see [3, 5]).

5.4 Reactivation of the tool provision and the exercise authoring phases

As a consequence of the deficiencies detected by the instructors during the *Exercise authoring* phase, we reactivated the *Tool provision* activity. We decided to change the constructivist approach followed by the first authoring tool to a generative one. Following this new approach, the instructors only had to specify the language processing task through an attribute grammar and a sentence in the language involved. Then, the new authoring tool was responsible for automatically generating the attributed syntax tree and the dependency graph among attributes necessary for the creation of the exercises. Therefore, we created an authoring tool that allowed the instructor to create and maintain languages defined by attribute grammars, which he/she can reuse, along with sentences in each language, to create an exercise [7, 38]. We also updated the assessment instruments related to the authoring tool by including aspects related to the tool's generative approach (in particular, the language and notation used to compose exercises).

Figure 4 shows the grammar editor view of the new authoring tool where the instructor can specify the language processor through an attribute grammar (in this case, the language processor defined in the tool corresponds to that of Fig. 1) and the exercise creator view (further explanations can be found in [5, 6]). When comparing the two authoring tools with respect to the same language processor (Fig. 1), we can see that with the tool following the generative approach instructors can create different exercises based on the same language effortlessly, as this tool directly generates the syntax tree and the dependency graph in the exercise from the language specification and a sentence in the language. This did not occur with the authoring tool based on the constructivist approach, with which the syntax tree of each sentence should be built.

After completing these changes, the *Exercise authoring* phase was reactivated. Instructors re-edited the collection of exercises using the new authoring tool, and in informal meetings with instructors we now obtained a very positive feedback about the new tool.

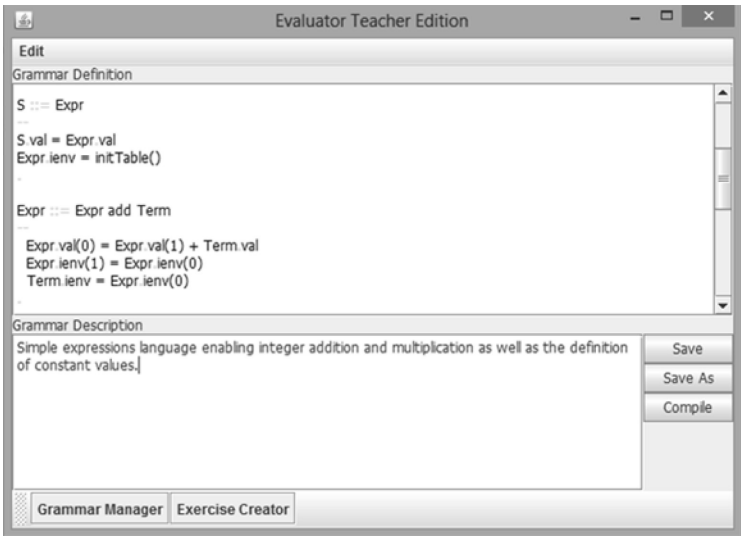


Fig. 4. Snapshots of the new *authoring tool*.

5.5 Simulation generation phase

Instructors used the simulator generator to group the exercises created in the previous phase and to generate serious games from meaningful groups of exercises. The game-based simulators generated transformed the collection of exercises into puzzle-based videogames where the syntax tree was transformed into a maze-world (each exercise represented a different level in the game; see [6] for more details). Inside this maze-world, the users could find boxes that correspond to attribute instances. The main goal of the game was to move the contents from these boxes to other boxes according to the attribute grammar of each exercise, in order to discover the contents of other boxes. The resulting game included *oracles*, represented as red statues with which the user could interact (Fig. 5d). Besides, the game provided some feedback associated to each box in the form of information panels where the user could see relevant information about the attribute that the box and its state represents (Fig. 5a). In addition, the user had access to an *inventory* to locate the attributes recorded in the maze (pointing to the room where the object was incorporated into the inventory, see Fig. 5c), and the student could consult the problem statement (Fig. 5b). For a more detailed description of the game-based simulators generated with *Evaluators* see [3, 6].

We created an assessment instrument for the resulting games in the form of a structured survey consisting of some Likert-like questions related to the learnability, interaction, usability, context adequacy and pedagogical utility of the educational games (these questions are those listed in Table 1). This Likert-like survey would be used in the exercise resolution phase. In addition, we held informal meetings with instructors in order to evaluate the simulator generator and the simulator assessment instruments themselves, and received very positive feedback.

5.6 Exercise resolution phase: experiences in the 2010–2011 and 2011–2012 academic years

The students first used the games generated in the 2010–2011 academic year. They participated in an experimental study oriented to assessing the educational effectiveness and adequacy of the educational artifacts and exercises:

- First, all the students participating answered a *pre-test* in order to gather information that was used to ensure the homogeneity of the two groups used in the study: the experimental group (students who used *Evaluators*) and the control group (students who followed a conventional paper-and-pencil method to solve the exercises).
- Once the groups were formed, the students solved *representative exercises*. Students in the experimental group solved the exercises within tutored sessions.
- Finally, all participants answered a *post-test*, designed to measure the degree of assimilation of the concepts and processes involved in the exercise resolution.

After analyzing the results of the post-test, we conducted a Shapiro-Wilk test in order to verify the normality of both samples. The test yielded enough significance to affirm that both groups can be adjusted to a normal distribution (p-value 0.108 for the control group and 0.681 for the experimental one). Then, we conducted a Levene test to verify the equality of variances between the two samples. The significance obtained from the test (0.819) shows

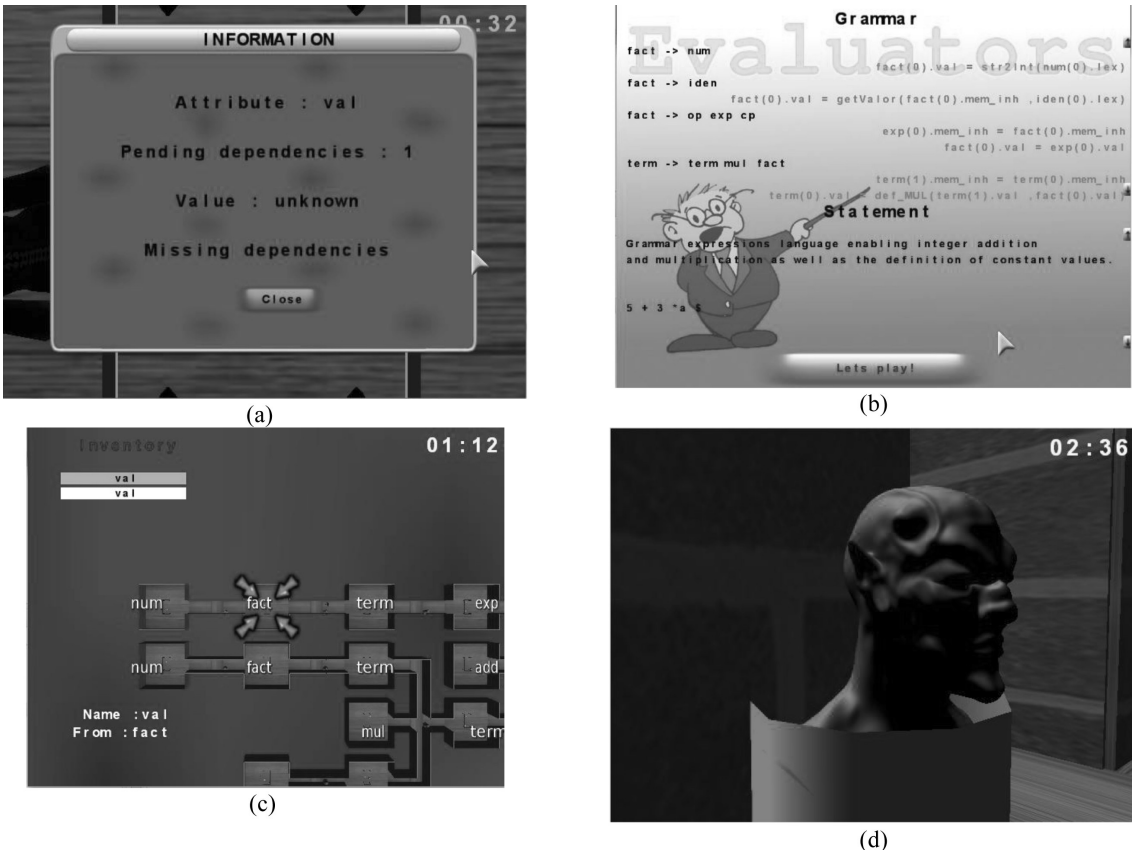


Fig. 5. Different screenshots of the features from the games generated.

that the variances are very similar. Finally, assuming that both groups had similar variances and they followed a normal distribution, we performed a t-student test in order to determine the equality of average hits of the two samples. The significance (0.78) obtained suggested that there were no significant differences between using *Evaluators* and following the traditional resolution method in the short term, which assumes that there is no decrease in quality between the method proposed and the traditional one in such a short time interval.

Likewise, we conducted the Likert-like survey devised in the previous phase on the experimental group (Table 1). Table 1 also presents the percentage of students who selected "Partly agree" or "Strongly agree" in their responses. The results show that *Evaluators* garnered popularity and success within this group of students as a complementary tool for the classes of compiler construction. Many students found it fun and useful to understand different aspects of the attribute grammars. It is interesting to note that students found *Evaluators* useful and, therefore, they would recommend it to their classmates. We also identified some aspects that should be improved. Clearly, the functionality of the oracles had to be improved. It was also necessary to clarify which attribute instance corresponded to which attribute copy of the inventory in order to reduce student errors.

Finally, we asked students for free opinions. On the one hand, students criticized some technical aspects of the games and pointed out some bugs. On the other hand, some of them proposed avoiding this kind of fun game-based simulators, which was a time-consuming task, and to adopt a more serious simulator based on visualizations.

Once the experiment was finished, we provided *Evaluators* to all the students and we advised them to use it in a self-regulated fashion in order to solve typical exercises concerning attribute grammars.

The experiment was repeated during the 2011–2012 academic year with similar outcomes (for the sake of concision we will omit the results here). Again, after the experiment, *Evaluators* was provided to all the students enrolled in the course. Once academic year 2011–2012 was finished, we looked for the effect of the comprehension method introduced by *Evaluators* in the long term performance of our students. As reported in [6], the results were very positive: the mean grades in years 2010–2011 and 2011–2012 increased by approximately one point with respect to the previous years (including the grades in the year 2009–2010, in which a problem-based intensive learning strategy was also adopted,

Table 1. Summary of the results obtained in the study of the degree of student satisfaction

Survey item	% agree
Pedagogical utility of the tool	
It helps me understand attribute grammar processing	73.7
It helps me understand the role of semantic attributes	73.7
It helps me understand the role of semantic equations	68.4
It helps me realize the differences among the different types of grammar attributes	73.7
I do not find it hard to realize my own mistakes when using the tool	36.8
Context adequacy	
Evaluators suitably complements lectures	84.2
Evaluators motivates me to study attribute grammar processing	52.6
Learnability	
I can rapidly start working with the tool without a long period of training	63.2
Mazes are a straightforward metaphor for representing attributed syntax trees	73.7
Using tables to collect information about attributes seems natural to me	52.6
Interaction	
I have used the problem statement several times	68.4
I have used the oracles	10.5
It is easy to identify which attribute instance corresponds to each attribute copy in the inventory	31.6
It is always easy to find out the state of problem solving	73.7
Overall usability judgment	
Evaluators is easy to use	42.1
I enjoyed using Evaluators	73.7
I would recommend Evaluators to other compiler construction students	68.4

but without obtaining such an improvement in the final performance).

5.7 Student assessment phase and reactivation of the tool provision phase

Once students completed the interactive game-like simulations in the 2010–2011 academic year, instructors used the analytic tool to examine the logs and evaluate the students' academic progress. Again, we assessed the adequacy of the analytic tool in informal interviews with the instructors. Instructors reported that the analytic tool, which only showed textual information about the content of the logs, needed to be improved. The instructors pointed to a more visual version of the tool that allowed them to see students' solutions at a glance.

In consequence we again reactivated the *Tool provision* phase, in this case to enhance the analytic tool. The new tool was able to display the syntax tree of the exercise proposed and it allowed instructors to navigate through the actions performed by the students. These actions are represented in the syntax tree as a flow of semantic attributes (see Fig. 6a). The new analytic tool, used to examine students' logs in the 2011–2012 experience, was welcomed by instructors [6].

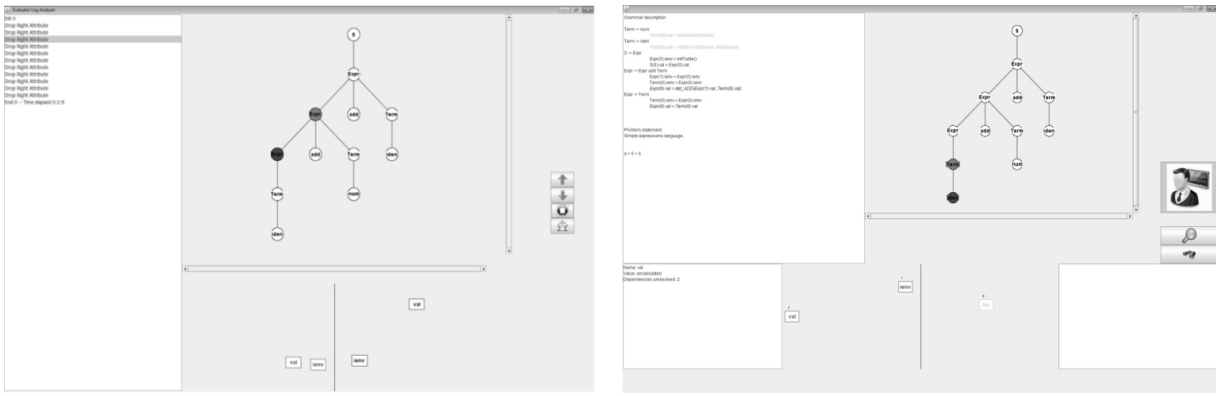
5.8 Rethinking the simulations

Considering the different reports obtained from students and instructors on the suitability of simulations based on serious games, we decided to reactivate the conception phase to remodel the type of simulation generated. Some students considered it difficult to solve the exercises in the simulations based on videogames, as they were distracted from the ultimate goal of the simulations due to the game aspect of such simulations. Thus, we decided to change the type of simulations generated by the system to a new one based on visualizations, where a representation of the syntax tree was displayed along with the other information considered in the simulation exercise. The aim of these new simulations was the same as the videogame based on student exploration of tree nodes to determine the order in which the attributes decorating such nodes were to be evaluated. Fig. 6b shows a snapshot of the new kind of simulation generated by the system. We also updated the simulation evaluation instrument in order to include aspects related to the newly designed kind of simulation.

Once the simulations were rethought, we restarted the *Tool provision* and *Simulation generation* phases in order to generate the new kind of simulations. During these phases, we obtained reports from different instructors which pointed to the suitability of the new type of simulations.

5.9 Exercise resolution phase: A new experience in the 2012–2013 academic year

During the 2012–2013 academic year students of the Compiler Construction course could use simulators based on visualizations to solve their exercises related to attribute grammars. In addition, we performed a new comparative study, recorded in the assessment instrument type of simulations, concerning the new kind of simulator generated against the traditional paper-and-pencil resolution method for the exercises contemplated. The experience



(a) (b)

Fig. 6. (a) snapshot of the new *analytic tool*, and (b) snapshot of the simulator based on visualizations generated by *Evaluators*.

Table 2. Agreement of students with survey items

Survey questions	% agree
The simulator has helped me better understand the processing model of attribute grammars.	78.57
The simulator has helped me better understand the role of semantic attributes during processing.	64.28
The simulator has helped me better understand what semantic equations do for processing.	85.71
The simulator has helped me better understand the differences between different types of attributes (lexical, synthesized, inherited).	50
When I made a mistake it made me realize that I was wrong.	42.85
The simulator conveniently complements the explanations given in lectures.	85.71
The simulator has encouraged my motivation to study the processing model of attribute grammars in general.	64.28

carried out is fully outlined in [4]. In this experience, again we divided the students into two different groups: those who used the simulators based on visualizations (experimental group) and those who did not use any software tool (control group).

In the first place, we performed an opinion survey among the students who used the simulators. The survey consisted of seven questions whose responses were scaled using a four-point scale based on the Likert-scale (*Totally disagree*, *Partly disagree*, *Partly agree*, *Totally agree*). Table 2 summarizes the results obtained from the survey, showing the percentage of students who totally or partially agreed with each survey item. The students positively valued the simulators as an adequate learning tool for the Compiler Construction course. Also, the feedback presented by the tool and its motivational component were perceived as key aspects for its success among students. However, some problems were detected by the students when they tried to identify their own mistakes, which should be solved in future improvements of the system.

Finally, as done with the serious game based simulator, we carried out a comparative study with the traditional method of resolution to measure the educational effectiveness of the newly designed simulator. We followed the same procedure described in Section 5.6, although we refined the nature of the post-test slightly (we used more in-depth questions involving practical skills concerning the use of attribute grammars). It also allowed us to refine the analysis method: instead of focusing on comparing mean grades, we focused on comparing distributions of students who passed and failed the post-tests. The results obtained are summarized in Fig. 7, which displays the count of students in each group who passed and failed the post-test. As can be seen at a glance, students who used simulations had a greater chance of successfully passing the final test.

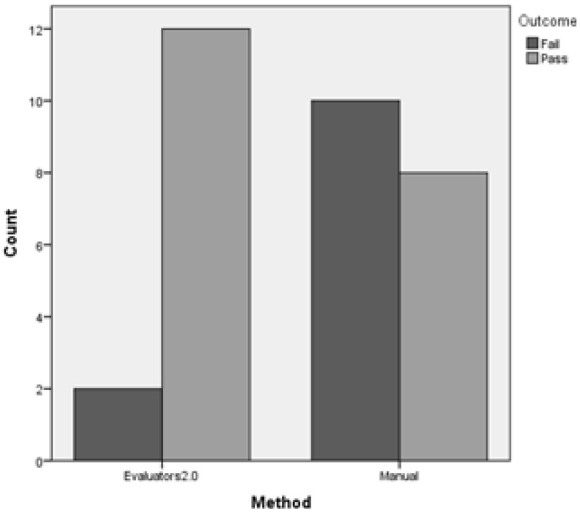


Fig. 7. Comparative bar chart of student performance.

In fact, this perceived difference is statistically significant (χ^2 test: p-value = 0.017).

So, we can conclude that this new version of *Evaluators* received a warm welcome by the students and showed promising effects in the students' performance.

6. Conclusions and future work

In this paper we have presented the development strategy followed in the construction of *Evaluators*, a system that uses interactive simulations as support for the learning of basic concepts in introductory compiler construction courses. The generative approach followed in *Evaluators* makes it possible to generate different kinds of simulations (based on serious games, based on more straightforward visualizations) from batteries of exercises. In addition, we introduced a continuous assessment process during the development of the system in order to promote the continuous enhancement of its different components. This assessment process not only drives the development strategy presented; it also provides useful information about instructors' and students' opinions regarding the software developed, and measures the educational effectiveness of the simulators generated. The results obtained, and presented in this paper, show that the different final software tools are well valued by students and instructors. Also, the effectiveness results registered evidenced that the simulators generated are useful to students. Encouraged by the results obtained in the development of *Evaluators*, we think that the development approach could be easily extrapolated to other domains in engineering education, where meaningful collections of exercises are usually used as a primary means of enabling the acquisition of basic engineering skills. For this purpose, the generative approach can make the active participation of instructors easier in the simulation development process, since their participation is limited to the provision of the collection of exercises (technical details concerning the production of the simulations are confined to the core of the generation software). In addition, continuous assessment is an essential element to ensure the final system meets the needs both of the instructors that provide the exercises and analyze students' performances, and the students playing the interactive simulations.

Nowadays, the reports obtained in the 2012–2013 academic year are being used to boost the *Conception* phase and remedy deficiencies identified, such as improving the feedback presented by the new type of simulator and including facilities to identify students' error in the exercise resolution. As immediate future work, we are planning to test these improvements in new experiences in the upcoming academic years. Likewise, we would like to compare the educational efficacy and efficiency of the two types of simulations generated by *Evaluators*, in order to establish the strengths and weaknesses of both simulators and find new ways to improve them. Finally, we are planning to extrapolate the development approach presented to other computer science and engineering education disciplines that greatly profit from the use of collections of exercises, such as Computer Programming or Web Applications for Mobile Devices.

Acknowledgments—Projects and grants TIN2010-21288-C02-01, TIN2009-13692-C03-03, Santander-UCM GR3/14 (group number 962022), and EDU/3445/2011 partially supported this research. Our thanks to Rafael Fernández-López and Ángel Valero-Picazo for their contribution to the first version of *Evaluators*.

References

1. A. A. Deshpande and S. H. Huang, Simulation games in engineering education: A state-of-the-art review, *Computer Applications in Engineering Education*, **19**(3), 2011, pp. 399–410.
2. W. K. Adams, S. Reid, R. LeMaster, S. B. McKagan, K. K. Perkins, M. Dubson and C. E. Wieman, A Study of Educational Simulations Part I—Engagement and Learning, *Journal of Interactive Learning Research*, **19**(3), 2008, pp. 397–419.
3. D. Rodríguez-Cerezo, M. Gómez-Albarrán and J. L. Sierra, From Collection of Exercises to Educational Games: A Process Model and a Case Study, *Proceedings of 11th IEEE International Conference on Advanced Learning Technologies (ICALT'11)*, 2011, pp. 282–284.
4. D. Rodríguez-Cerezo, M. Gómez-Albarrán and J. L. Sierra, Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts, *Proceedings of 18th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2013)*, 2013, pp. 28–33.
5. D. Rodríguez-Cerezo, A. Sarasa, M. Gómez-Albarrán, and J. L. Sierra, Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game Based Approach, *Proceedings of International Simposium on Computers in Education (SIIE'12)*, 2012, pp. 1–6.
6. D. Rodríguez-Cerezo, A. Sarasa-Cabezuelo, M. Gómez-Albarrán and J. L. Sierra, Serious games in tertiary education: A case study concerning the comprehension of basic concepts in computer language implementation courses, *Computers in Human Behavior*, **31**(Feb.), 2014, pp. 558–570.
7. J. Paakki, Attribute Grammar Paradigms—A High-Level Methodology in Language Implementation, *ACM Computer Surveys*, **27**(2), 1995, pp. 196–255.
8. D. Rodríguez-Cerezo, M. Gómez-Albarrán and J. L. Sierra, A process model for the generative production of interactive simulations in engineering education, *Proceedings of First International Conference on Technological Ecosystem for Enhancing Multiculturality*, 2013, pp. 95–103.
9. A. Sarasa-Cabezuelo and J. L. Sierra-Rodríguez, Software engineering for eLearning, *Proceedings of the First International Conference on Technological Ecosystem for Enhancing Multiculturality*, 2013, pp. 81–86.
10. K. Friesen and H. Schmitz, Managing the development of an e-learning product—Applying software engineering techniques in an academic environment, *Proceedings of the World Congress of Networked Learning in a Global Environment*, 2002, pp. 40–46.
11. S. Marshall and G. Mitchell, Applying SPICE to e-learning: an e-learning maturity model? *Proceedings of Sixth Austra-*

lasian Conference on Computing Education, 2004, pp. 185–191.
12. W. C. Allen, Overview and evolution of the ADDIE training system, *Advances in Developing Human Resources*, **8**(4), 2006, pp. 430–441.
13. S. Hambach and A. Martens, ROME: Systems Engineering in Technology Enhanced Learning, *Proceedings of Eighth IEEE International Conference on Advanced Learning Technologies*, 2008, pp. 733–734.
14. F. J. Lage, Y. Zubenko and A. Cataldi, An extended methodology for educational software design: some critical points, *Proceedings of the 31st Annual Conference Frontiers in Education*, 2001, pp. 13–18.
15. G. Paquette, I. Rosca, I. De la Teja, M. Léonard and K. Lundgren-Cayrol, Web-based Support for the Instructional Engineering of E-learning Systems, *Proceedings of World Conference of the WWW and Internet (WebNet'01)*, 2001, pp. 981–987.
16. A. P. Costa, M. J. Loureiro and L. P. Reis, Development methodologies for educational software: the practical case of courseware SERe, *Proceedings of Conference on Education and New Learning Technologies*, 2009, pp. 5816–5825.
17. I. García-Pacheco and J. García-Matías, A Methodology Based on Effective Practices to Develop Educational Software. *Computación y Sistemas*, **11**(4), 2008, pp. 313–332.
18. C. González, P. Toledo, V. Muñoz, M. A. Noda, A. Bruno and L. Moreno, Inclusive educational software design with agile approach, *Proceedings of 1st International Conference on Technological Ecosystem for Enhancing Multiculturality*, 2013, pp. 149–155.
19. P. Avgeriou, S. Retalis and N. Papaspyrou, Modeling learning technology systems as business systems, *Journal of Software and Systems Modeling*, **2**(2), 2003, pp. 120–133.
20. D. Díez, C. Fernández and J. Dodero, A systems engineering analysis method for the development of reusable computer-supported learning systems, *Interdisciplinary Journal of E-Learning and Learning Objects*, **4**(1), 2008, pp. 243–257.
21. S. De Freitas and S. Jarvis, Towards a development approach for serious games, *Games-based learning advancements for multi-sensory human-computer interfaces: Techniques and effective practices*, 2008, Hershey, PA: IGI Global.
22. K. Kiili, Digital game-based learning: Towards an experiential gaming model, *The Internet and higher education*, **8**(1), 2005, pp. 13–24.
23. R. J. Nadolski, H. G. Hummel, H. J. Van Den Brink, R. E. Hoefakker, A. Slootmaker, H. J. Kurvers and J. Storm, EMERGO: A methodology and toolkit for developing serious games in higher education, *Simulation & Gaming*, **39**(3), 2008, pp. 338–352.
24. M. A. Gómez-Martín, P. P. Gómez-Martín and P. A. González-Calero, Dynamic Binding is the Name of the Game, *Proceedings of International Conference on Entertainment Computing (ICEC'06)*, 2006, pp. 229–232.
25. P. Moreno-Ger, I. Martínez-Ortiz, J. L. Sierra and B. Fernández-Manjón, A Content-Centric Development Process Model, *IEEE Computer*, **41**(3), 2008, pp. 24–30.
26. P. Moreno-Ger, J. L. Sierra, I. Martínez-Ortiz and B. Fernández-Manjón, A documental approach to adventure game development, *Science of Computer Programming*, **67**(1), 2007, pp. 3–31.
27. A. C. Vidani and L. Chittaro, Using a task modeling formalism in the design of serious games for emergency medical procedures, *Proceedings of Games and Virtual Worlds for Serious Applications* (VS-GAMES'09), 2009, pp. 95–102.
28. T. Boyle, Design principles for authoring dynamic, reusable learning objects, *Australian Journal of Educational Technology*, **19**(1), 2003, pp. 46–58.
29. C. Bradley and T. Boyle, The design, development and use of multimedia learning objects, *Journal of Educational Multimedia and Hypermedia (Special Edition on Learning Objects)*, **13**(4), 2004, pp. 371–389.
30. V. Štuikys and R. Damaševicius, Towards knowledge-based generative learning objects, *Information Technology and Control*, **36**(2), 2007, pp. 202–212.
31. V. Štuikys and R. Damaševicius, Development of generative learning objects using feature diagrams and generative techniques, *Informatics in Education*, **7**(2), 2008, pp. 277–288.
32. E. J. R. Koper and J. M. Manderveld, Educational modelling language: modelling reusable, interoperable, rich and personalised units of learning, *British Journal of Educational Technology*, **35**(5), 2004, pp. 537–552.
33. I. Martínez-Ortiz, J. L. Sierra, B. Fernández-Manjón and A. Fernández-Valmayor, Language Engineering Techniques for the Development of e-Learning Applications, *Journal of Network and Computer Applications*, **32**(5), 2009, pp. 1092–1105.
34. S. Diehl, A. Kerren, and T. Weller, Visual Exploration of Generation Algorithms for Finite Automata on the Web, *Lecture Notes in Computer Science: Implementation and Application of Automata*, 2001, pp. 327–328.
35. P. A. Santos and J. J. Castro-Sánchez, Una herramienta para la enseñanza y aprendizaje de la asignatura Procesadores de Lenguajes, *Proceedings of XII Jornadas de la Enseñanza Universitaria de la Informática*, 2006, pp. 499–507.
36. J. L. Sierra, A. M. Fernández-Pampillón and Fernández A. Valmayor, An environment for supporting active learning in courses on language processing, *Proceedings of 13th annual conference on Innovation and technology in computer science education (ITiCSE'08)*, 2008, pp. 128–132.
37. D. R. Sadler, Formative Assessment and the Development of Instructional Systems, *Instructional Science*, **18**(12), 1989, pp. 119–144.
38. A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, *Compilers: principles, techniques and tools (second edition)*, 2007, Addison-Wesley.

Daniel Rodríguez-Cerezo is a PhD student and works at the university as a doctoral fellow at the Software Engineering and Artificial Intelligent Department of the Complutense University of Madrid (UCM), Spain. He is a member of the research group ILSA (Implementation of Language-Driven Software and Applications: http://ilsa.fdi.ucm.es). His research is focused on the development of software tools driven by domain specific languages in the fields of e-Learning (videogames, simulators and prototyping tools), and on educational repositories for teaching design and implementation of computer languages. Besides, he is interested in the development and improvement of software language engineering techniques.

Antonio Sarasa-Cabezuelo is a full-time Lecturer in the Computer Science School at the UCM. His research is focused on the language-oriented development of XML-processing applications, and on the development of applications in the fields of digital humanities and e-Learning. He was one of the developers of the Agrega project on digital repositories (a pioneer project in this field in Spain). He is a member of the research group ILSA. He has participated in several research projects in the fields of software language engineering, digital humanities and e-learning, and he has published over 50 research papers in national and international conferences.

Mercedes Gómez-Albarrán is an associate professor in the Department of Software Engineering and Artificial Intelligence at the UCM. She received the Ph.D. degree in Computer Science from UCM in 2000. In the last 8 years, she has been Vice Dean of the Computer Science School, UCM. She is a member of the Group of Artificial Intelligence Applications (GAIA)

at UCM. Her research interests cover the virtual learning environments and the serious games, and also the knowledge-based recommendation techniques and their application to e-learning, with over 60 scientific publications.

José-Luis Sierra is an associate professor at the UCM's Computer Science School, where he leads the ILSA Research Group. His research is focused on the development and practical uses of computer language description tools and on the language-oriented development of interactive and web applications in the fields of digital humanities and e-Learning. Prof. Sierra has leaded, and participated in, several research projects in the fields of digital humanities, e-Learning and software language engineering, the results of which have been published in over 100 research papers in international journals, conferences and book chapters. He serves regularly as reviewer / PC Member for several international reputed journals and conferences.

6.3 A process model for the generative production of interactive simulations in engineering education

Cita completa:

Rodríguez-Cerezo, D., Gómez-Albarrán, M., Sierra, J. L. A Process Model for the Generative Production of Interactive Simulations in Engineering Education. En TEEM'13 : Proceedings of the First International Conference on Technological Ecosystem for Enhancing Multiculturality, 95-103. 2013.

Resumen original de la contribución:

This paper introduces a process model for the production of interactive simulations that promotes the automatic generation of this kind of artifacts from representative collections of exercises. This approach is especially well suited for the early stages of Engineering Education, due to the importance played by interactive simulations and the resolution of carefully-designed batteries of exercises in the acquisition of basic skills in engineering disciplines. According to this process model, instructors are equipped with: (i) an authoring tool, which helps them to provide and maintain the collections of exercises, (ii) a generator, able to automatically turn collections of exercises into executable interactive simulators, and (iii) an analytic tool that can be applied to the logs produced by such simulators in order to assess student performance. In addition, the model promotes the improvement of the resulting educational systems through an exhaustive and continuous evaluation process. Besides to detail the process model, this paper illustrates it with the development of Evaluators, an educational system for the generation of different kinds of interactive simulations in introductory Compiler Construction courses typically taught in Computer Science and Computer Engineering degrees.

Referencia de citas bibliográficas:

[5][3][9][32][35][37][41][42][55][58][75][93][111][112][120][136][139][141] [142][146][158]

A Process Model for the Generative Production of Interactive Simulations in Engineering Education

Daniel Rodríguez-Cerezo
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9 28040 Madrid (Spain)
(+34) 91 3947606
drcerezo@fdi.ucm.es

Mercedes Gómez-Albarrán
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9 28040 Madrid (Spain)
(+34) 91 3947561
albarran@sip.ucm.es

José-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9
28040 Madrid (Spain)
(+34) 91 3947548
jlsierra@fdi.ucm.es

ABSTRACT

This paper introduces a process model for the production of interactive simulations that promotes the automatic generation of this kind of artifacts from representative collections of exercises. This approach is especially well suited for the early stages of Engineering Education, due to the importance played by interactive simulations and the resolution of carefully-designed batteries of exercises in the acquisition of basic skills in engineering disciplines. According to this process model, instructors are equipped with: (i) an *authoring tool*, which helps them to provide and maintain the collections of exercises, (ii) a *generator*, able to automatically turn collections of exercises into executable interactive simulators, and (iii) an *analytic tool* that can be applied to the logs produced by such simulators in order to assess student performance. In addition, the model promotes the improvement of the resulting educational systems through an exhaustive and continuous evaluation process. Besides to detail the process model, this paper illustrates it with the development of *Evaluators*, an educational system for the generation of different kinds of interactive simulations in introductory Compiler Construction courses typically taught in Computer Science and Computer Engineering degrees.

Categories and Subject Descriptors

D.2.9 **[Software Engineering]**: Management – *software process model*.

D.1.2 [**Programming Techniques**]: Automatic Programming.

K.3.1 [**Computers and Education**]: Computer Issues in Education – *computer-assisted Instruction (CAI)*.

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*.

General Terms

Design, Management, Human Factors, Languages.

Keywords

Development Process Model, Application Generators, Computer Science and Computer Engineering Education, Serious Games, Interactive Simulations, Attribute Grammars.

TEEM '13, November 14 - 15 2013, Salamanca, Spain
Copyright 2013 ACM 978-1-4503-2345-1/13/11… $15.00
http://dx.doi.org/10.1145/2536536.2536552

1. INTRODUCTION

Interactive simulations have proven to be powerful tools for learning and have focused principally on teaching and practicing mental processes rather than concepts. Additionally, interactive simulations motivate learners by involving them in attractive experiences [1]. For these reasons, interactive simulations are frequently used in Engineering Education settings [7]. Building a successful interactive simulation is not an easy task, however, and requires the effective collaboration of different types of actors that master necessary skills or knowledge. The combination of these skills lets development teams create the simulator. Therefore, in order to bridge the collaboration between the actors it is mandatory to propose effective procedures to norm the form in which instructors and developers collaborate during the development of this kind of artifacts. Additionally, it is very important to continuously assess the production process, and to actively involve students in the assessment of the final product.

In this paper we address this question by giving a process model that makes a strong use of generative principles and uses the exercises proposed by the instructors in the particular learning domain as a keystone element on which to base the overall life-cycle. Since Engineering Education is traditionally supported by carefully designed collections of exercises oriented to develop particular abilities of the day-to-day engineering practice, to base the development of interactive simulators in these batteries of exercises is a meaningful choice in this kind of settings. Thus, in the process model proposed the structure of exercises is a primary element that is used: (i) to provide suitable authoring tools for describing the exercises and their solutions, (ii) to propose adequate metaphors that reformulate the process of solving exercises as the process of playing interactive simulations, and (iii) to devise generators able to automatically generate such simulators from the collection of exercises. In order to illustrate this process model, we will focus on the development of *Evaluators* [16][17][18][19], an interactive-simulation-oriented educational system that promotes the learning of fundamental concepts in introductory Compiler Construction courses (in particular, concepts related to syntax-directed translation and attribute grammars [2][15]).

The rest of the paper is organized as follows: section 2 outlines some works related to ours; section 3 describes the proposed process model; section 4 describes how this process model was used to develop *Evaluators*; section 5 compares our proposal with the related work; and finally, section 6 provides final conclusions and some lines of future work.

2. RELATED WORK

The process model presented in this paper is based on our previous work on the production of learning contents and tools in Computer Science and Engineering disciplines [11][21]. A preliminary version of this process model can be found in [16].

There are several methodologies for developing educational systems. For instance, ADDIE (Analysis – Design – Development – Implementation – Evaluation), one of the most widespread instructional system design methodologies, puts a strong emphasis on the organizational aspects of the instructional design process and can be applied not only to technology-enhanced but also to conventional learning systems [3]. Specially targeted to the development of educational software, the methodology described in [12] integrates phases ranging from the feasibility of the tool to the withdrawal or maintenance of the software. They pass through other phases as prototype design and development or testing. Also, the author give different advises and tips in order to organize the work team, the documentation associated to the different stages of the methodology or the evaluation of the software. The work described in [9] proposes a more pragmatic approach based on a repertory of good (*effective*) practices. Finally, the work in [5] proposes a development methodology based on user-centered and agile approaches.

Concerning more specific domains, in particular educational games, the work described in [14] proposes a process model for the language-driven development of educational videogames, while the described in [10] proposes to articulate the production of educational games on a very flexible framework that facilitates the procedural customization of different games in different domains using, among another elements, exercises provided by instructors. Concerning generative approaches, in [6][13] the concept of *generative learning object* is introduced as a way of parameterizing learning objects with a set of parameters, which can be subsequently set in order to customize the learning materials to the need of the particular learners. In [4] a language-driven approach for the development of e-learning applications is described, according to which e-learning components are conceived as language processors for formal languages in which students can describe their solutions, therefore allowing the components to check whether those solutions are correct and to give appropriate feedback to the students. Other examples in the field of the case-study presented in this work are GANIFA [8], a generative visualization tool that allows the student to visualize and animate finite automata, or PAG [21], a generative tool for the creation of visualizations of attributed syntax trees very similar to the *Evaluators* authoring tool.

3. THE GENERATIVE DEVELOPMENT PROCESS MODEL

The keystone aspect of our process model is to encourage a *generative* approach, according to which interactive simulations are generated from representative collections of exercises proposed by the instructors. It means that the instructors provide learning content, in the form of exercise statements and their solutions, in order to generate the interactive simulations that the students will use. In this generative approach the instructors control the final steps of the simulations' production. This approach grants certain level of customization on the educational artefacts generated without the assistance of a development team.

Next, we dissect this process model from two different perspectives. First, we describe the products and activities involved by the generative development (subsection 3.1). Finally, we describe how a continuous assessment fits in the process model as an aspect orthogonal to each activity (subsection 3.2).

3.1 Products and activities

The products and activities involved in our production process model are sketched in Figure 1:

- The production process starts with the *Conception* activity. The objective of this activity is to provide the domain models that will drive the rest of the process. On the one hand, it is needed to provide a model of the exercises that will let students strengthen and go deeply into the concepts taught in class: the *exercise model*. This model will characterize both the exercise statements and solutions. On the other hand, it is needed to model interactive simulations based on the mental processes required to solving this kind of exercises. The results are gathered in the *simulator model*. This model will be largely based on a suitable metaphor allowing the transformation of collections of exercises and their solutions (i.e., instances of the exercise model) into running interactive simulations. This activity involves both *instructors*, since they know the learning domain and they are able to identify the right nature of exercises and metaphors to transform them into interactive simulators, and *developers*, since they have the necessary background of Computer Science required to express the models with the required level of detail.
- The following activity is *Tool provision*. This activity is oriented to provide the toolset required to successfully produce and exploit the interactive simulations: the *authoring tool*, the *simulator generator* and the *analytic tool*. Being it an activity with a strong software development component, it is mainly carried out by *developers*. However, *instructors* also participate in order to support developers on the peculiarities of the domain models.
- Once the tools are available, during the *Exercise authoring* activity instructors create collection of *exercises* using the authoring tool. Indeed, this tool is conceived as a specialized editor that helps instructors to describe the exercises that integrate the collections, and which is able to automatically transform such descriptions into suitable instances of the *exercise model*.
- Then, once the collection of exercises is available, the interactive simulations are actually produced in the *Simulator generation* activity. For this purpose, instructors use the *simulator generator*. This generator typically works by transforming the exercise model instance into a suitable instance of the simulator model. This transformation will be based on the aforementioned metaphor. Then, depending on the organization paradigm used, this instance can: (i) be interpreted by a suitable player, (ii) point to behavior required for its execution (if this behavior was attached as methods of the classes of the simulator model), etc. In any case the result of this activity will be the running interactive simulation.
- Interactive simulations will be used by students in order to solve the exercises during the *Exercise resolution* activity. In addition, the process model supports the production of simulations that records student behavior in significant *logs*. Finally, instructors can exploit the *logs* produced during the exercise resolution to assess student performance. It is done in the *Student assessment* activity, and it is facilitated by the use of the *analytic tool*. By exploiting the logs produced by

the interactive simulation in the *Exercise resolution* activity, the tool helps instructors to study the students' behaviour after using the simulation in order to assess the progress of the students and identify the leaks and problems they found. The result of this analysis is registered in the *grades* report, in which instructors adequately grade students.

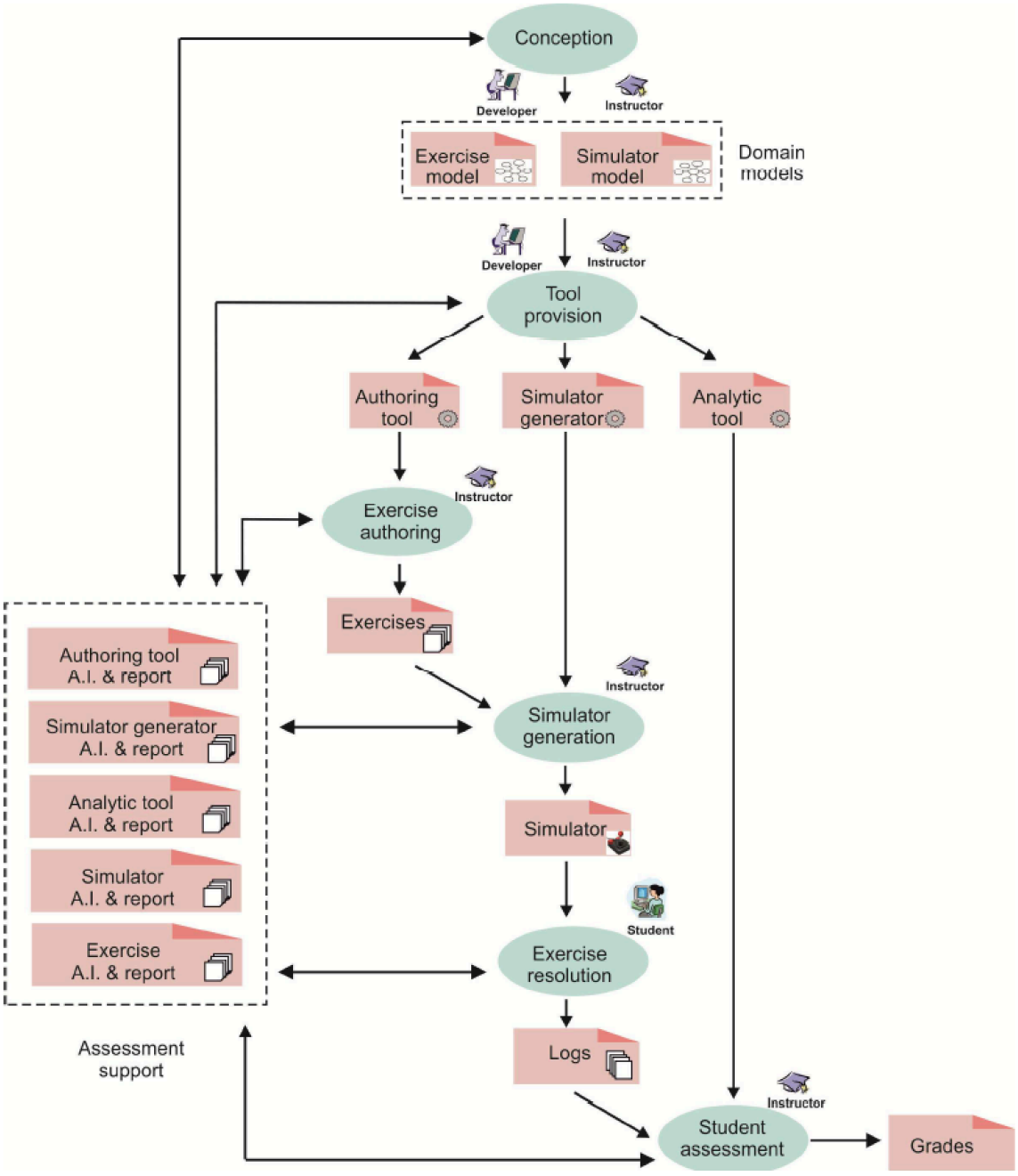


Figure 1. Products, activities and participants of the process model

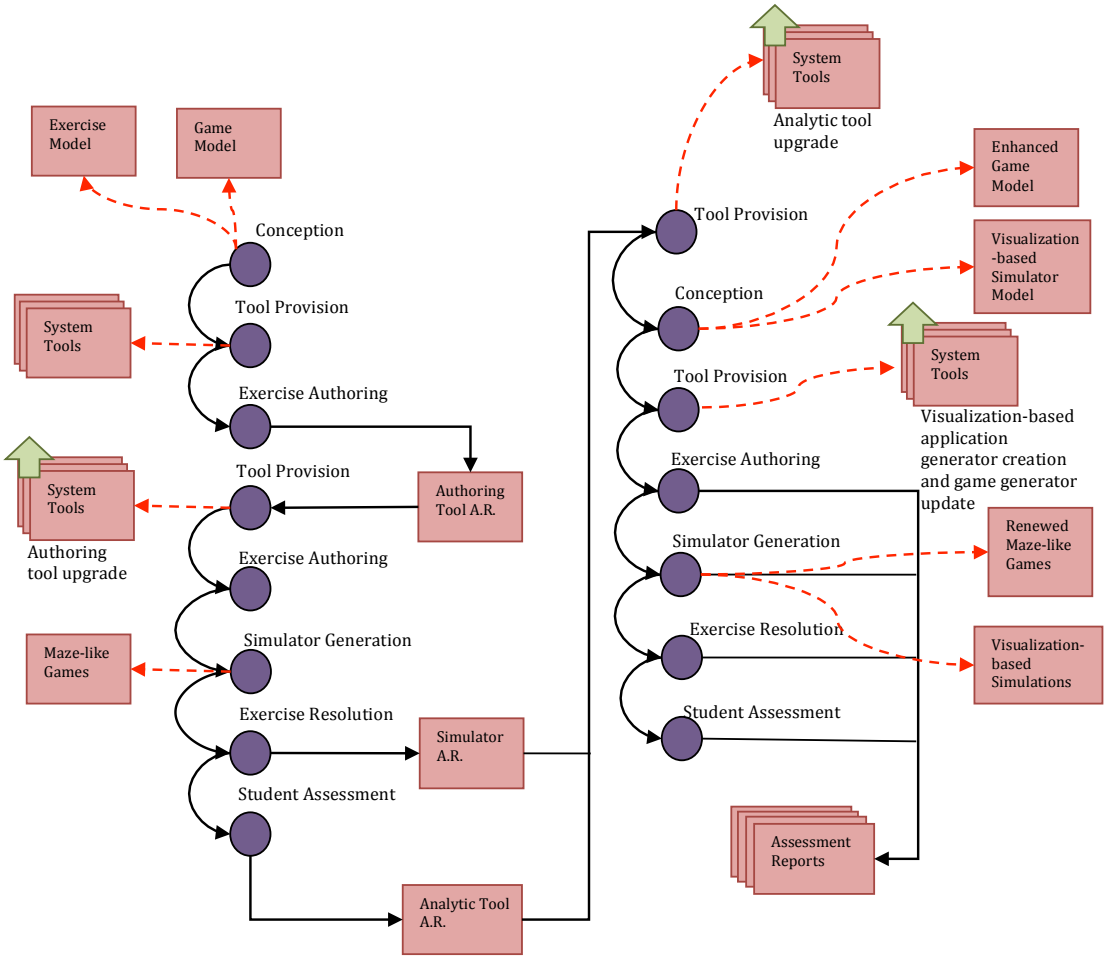


Figure 2 Process model activities executed during the development of *Evaluators*

3.2 Continuous Assessment

In addition to rule the pure production of software assets, our process model makes a strong emphasis on continuous formative assessment. Being formative, this assessment is oriented to improve the different aspects of the resulting educational systems [20] (in our case, domain models, tools and generated interactive simulations). For this purpose, assessment is considered as an orthogonal aspect to all the activities of the process model. In addition, on the basis of the evaluation results, production can proceed along several iterations, each one leading to products better suited to the needs of instructors and students.

In each activity assessment is guided by two main assets:

> *Assessment Instruments*. These assets (indicated by A.I in Figure 1) capture all the information required to carry out the assessment process. Typical examples of these instruments are surveys and questionnaires to be filled by instructors and students, descriptions of statistical procedures to measure educational efficacy of interactive simulations, etc. *Assessment Reports*. These assets record the results obtained from the application of the assessment instruments to the corresponding activities and are eventually used to restart the corresponding activities in order to overcome the problems detected and improve the resulting products.

Next subsections detail the different assessment instruments and reports involved in the process model, and how the outcome of these reports affect to the normal flow of the production process by reactivating activities with the aim of improving the overall educational system.

Assessment instruments

The assessment instruments contains: (i) detailed descriptions of the procedures required to evaluate the different aspects related to a concrete product in the production process (e.g., a description of how to carry out a survey, a description concerning an experimental design, etc.), and (ii) the additional information required by such procedures (e.g., the concrete questions that makes up the survey, the concrete parameters to apply to the experimental design, etc.). In particular, the process model introduces five categories of assessment instruments, one for each concrete product envisioned:

- There is one assessment instrument to evaluate the different aspects of each tool produced in the *Tool provision* activity (i.e., *authoring tool*, *simulator generator* and *analytic tool*). Each instrument will enable the evaluation of aspects like quality, usability and adequacy to the instructors' needs of the corresponding tool. These instruments are designed by developers and are used by instructors.
- There is another instrument to evaluate the educational adequacy and effectiveness of the particular collection of exercises produced by using the *authoring tool*. These instruments are designed by instructors and they can be used both by instructors and students.
- Finally, there is a fifth instrument oriented to assess the adequacy of the interactive simulation. This instrument will evaluate aspects like usability, accessibility, educational effectiveness, etc. This instrument is collaboratively designed by developers and instructors, and it is mainly targeted to students.

It is worthwhile to notice that the exact nature of each instrument will depend of the particular development. In addition, it is important to remark that the assessment instruments are focused on concrete products in order to let non-technical participants (instructors and students) carry out the actual assessment. In this way, the underlying domain models are implicitly assessed through the different tools produced and the simulations generated.

Assessment reports

As a consequence of applying the assessment instruments, it is possible to obtain different assessment evidences which are recorded in the assessment reports. The analysis of these reports can reactivate some activities in order to solve the deficiencies detected. Therefore, as indicated earlier, the process model encourages a continuous enhancement based on the collected assessment evidences. This enhancement does not only affect to products obtained during the normal production flow (subsection 2.1), but it also can imply the enhancement of the assessment instruments themselves.

Thus, the process model envisions a different report for each assessment instrument conceived:

- The *authoring tool*, *simulator generator* and *analytic tool* reports will contain the assessment evidence gathered by instructors as result of applying the corresponding tool assessment instruments. Depending of the assessment outcome, the analysis of this report could imply to reactivate the *Conception* activity (in order to improve the domain models) and/or the *Tool provision* activity (in order to improve some technical aspects of the tools).
- The *exercise* report will describe the outcomes of assessing the collection of exercises. It could imply to restart the *Exercise authoring* activity (for improving some aspects of the exercises), but also the *Conception* activity (in order to extend the domain models for covering new aspects not covered by the current ones) and, as a consequence, the *Tool provision* activity (in order to mirror the extensions of the models in the tools).
- Finally, the *simulator* report will contain evaluation evidences concerning the interactive simulation. As a consequence, it could restart any of the upper activities: *Simulator generation* (to better fix some generation parameters), *Exercise authoring* (to increase the exercise quality, number, etc.), *Tool provision* (to enhance some aspects of the generation process and/or the underlying generic software supporting it), and even *Conception* (to enhance domain models) which would be followed by the consequent reactivation of the *Tool provision* (to propagate the changes to the corresponding tools).

Finally, we should remark again that, in addition to the previously mentioned enhancements, assessment instruments could be also object of enhancement anytime in the production process.

4. CASE STUDY: *EVALUATORS*

In order to illustrate how our process model can be applied in practice we will describe the development of *Evaluators* [16][17][18][19], an educational system for the generation of two different kinds of interactive simulations.

The aim of *Evaluators* is the generation of interactive simulations oriented to enhance the learning and teaching of fundamental concepts of introductory Compiler Construction courses. In particular, the system is focused on enhancing the teaching and learning of basic concepts on *syntax-directed translation*. For this purpose, the system adopts the attribute grammars formalism [15] as the main paradigm for undertaking syntax-directed translation. The system has been used during the last three academic years at Complutense University of Madrid (Spain).

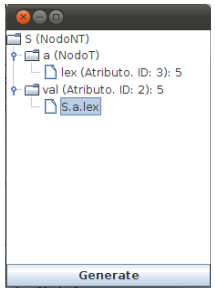
Figure 2 summarizes the dynamics of *Evaluators* development according to our process model[13]. For this purpose, the activation and re-activation of the different activities are highlighted. In this way:

- The development of *Evaluators* began during the 2009-2010 academic year, by setting the *exercise* and *simulator* models during a first activation of the *Conception* activity. In this first conception round, exercises were based on syntax-directed translation tasks defined by: (i) an informal and formal (attribute grammar) description of a language processing tasks, (ii) a sentence of the language to be processed, and (iii) the attributed syntax tree built by processing the sentence according to the attribute grammar. The students had to provide, as a solution, a valid evaluation order of the attributes that decorated the syntax tree. On the other hand, the *simulator model* was set as a game-based system where the syntax trees of the exercises were transformed into a maze-like world. In this world attributes were represented by boxes. The aim of the game was to move these boxes through the maze, in order to emulate the evaluation process (see [18][19] for more details).
- Then, during the *Tool provision* activity, we developed each software tool needed: *authoring tool*, *simulation generator* and *analytic tool*. The first version of the *authoring tool* consisted in a visual editor where the instructor could specify the statement of the exercises (informal and formal description of the syntax-directed processing task), and also to edit the corresponding

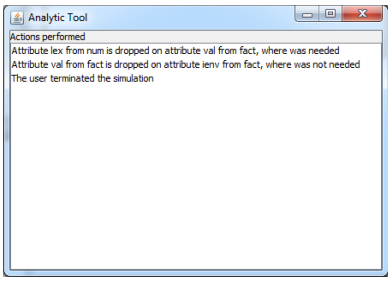
[13] The *assessment instruments* are created and updated accordingly when creating and upgrading their corresponding products. In order to simplify Figure 2 and the text of this section, references to the *assessment instruments*' creation and update are omitted.

attributed syntax tree [18]. This tree contained both the attribute values and the dependencies among these values, therefore representing all the constraints to be satisfied by the feasible solutions [2][15]. In this way, instructors were compelled to create the tree nodes and interconnect them. They should then create instances of each attribute and associate these instances with the corresponding tree nodes. Finally, they should link the instances of attributes to create the dependency graphs constraining the possible evaluation orders. Figure 3a shows an snapshot of this *authoring tool*. On the other hand, the *analytic tool* was confined to show a basic list of the actions carried out by the students (see Figure 3b). Finally, the *simulation generator* operated by transforming the exercises in descriptions of the maze-like games, following the metaphor outlined above.

•



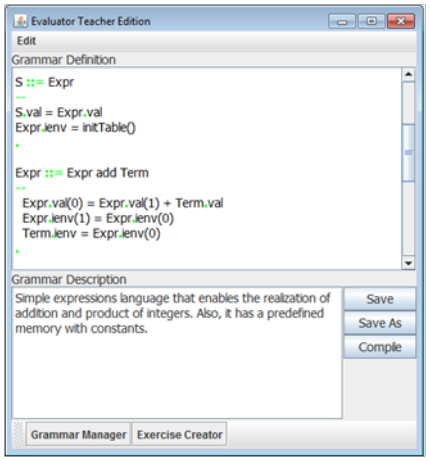
(a)



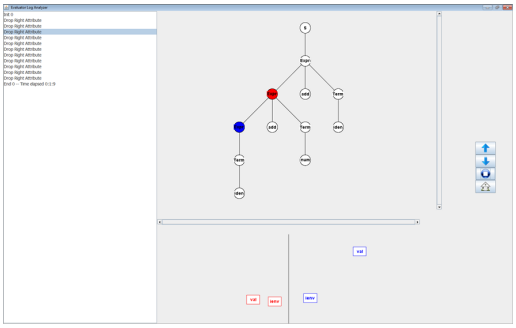
(b)

Figure 3 (a) Snapshot of the first *authoring tool*; (b) Snapshot of the *analytic tool*

- During the *Exercise provision* activity, firstly carried out during the summer of 2010, the instructors made a first attempt of using the *authoring tool* to design a battery of exercises (actually, to encode an existing battery of exercises in terms of the *exercise model*). Assessment in this activity was carried out using informal interviews with the instructors. These interviews revealed a serious usability leak in the *authoring tool*: the need of editing the attributed syntax tree, which resulted in a tedious and error-prone process.
- As a consequence of the deficiencies in the *authoring tool*, we reactivated the *Tool provision* activity and overcame the problems detected. The new *authoring tool* incorporated a generative component able to automatically generate the attributed syntax tree (i.e., the constraints on the feasible solutions) from the attribute grammar description (it was formally given using a suitable domain-specific language close to the used by instructors in their classrooms [2][15]). Figure 4a shows a snapshot of the new *authoring tool*.



(a)



(b)

Figure 4 Snapshots of the new *authoring tool* (a) and the new *analytic tool* (b)

- Using the new *authoring tool* the exercises were successfully created and we obtained a very positive feedback from instructors about its usability and adequacy (informal interviews were also used) [19]. Then, instructors proceeded to generate the interactive simulations from these exercises as maze-like videogames (see Figure 5a).
- Then, in the 2010-2011 academic year, the students used the videogames generated (*Exercise resolution* activity). As assessment instrument for assessing the games we used a structured survey made of different Likert-like items and oriented to assess aspects like usability, adequacy to the learning goals, etc. [19]. Once the surveys were filled (and, therefore, the corresponding assessment reports produced), several leaks in the generated games were revealed, mainly concerning their usability.
- Once students completed the interactive simulations, instructors proceeded to examine the logs generated in

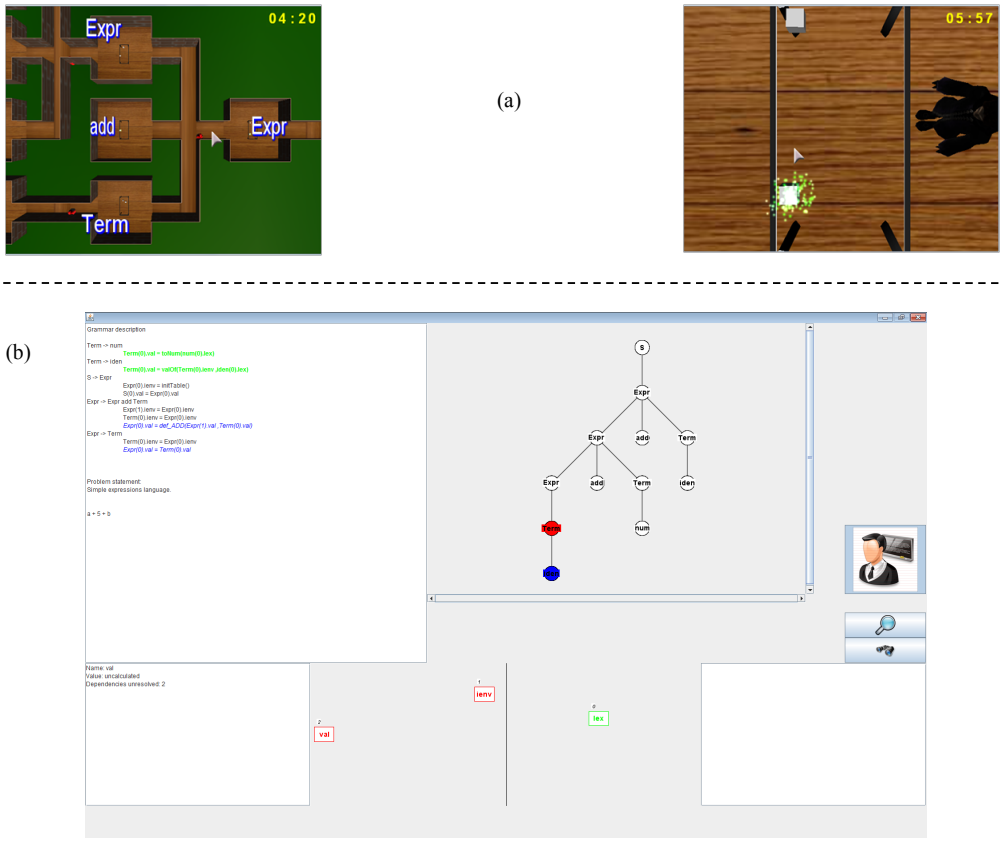


Figure 5 (a) Captures of the simulators based on videogames and (b) based on visualizations generated by *Evaluator*

order to assess their performance. Again we used informal interviews to assess the adequacy of the *analytic tool*. As a consequence we realize that the simple tool used was not enough to facilitate the assessment of the students. Therefore, during summer of 2011 we reactivated again the *Tool provision* activity to enhance this tool. The new tool was able to display the behavior of the student directly on the attributed syntax tree (see Figure 4b), and this time was welcomed by instructors as a useful analytic artifact [19].

- Thus, in the 2011-2012 academic course we got a stable and fully operative version of *Evaluators*. As consequence of the reports on the interactive simulations collected among students during the 2010-2011 and 2011-2012 academic years, in summer of 2012 we reactivated the *Conception* activity. On the one hand, some enhancements to the game look and dynamics were suggested. On the other hand, a not inconsiderable number of students pointed that, although compelling, the game component of the simulations distracted them from the main objective of solving the proposed exercises. In order to provide a more suitable help to this type of students, we decide to additionally provide an alternative *simulator model* based on a more straightforward simulation (visualization-based simulations), more oriented to direct manipulation of the attributed syntax trees [17] (see Figure 5b). Consequently, the *Tool provision* activity was also reactivated in order to update the *simulator generator* of the games and to develop a new *simulator generator* for the second type of educational applications that can be generated within *Evaluators*.

- Finally, during the 2012-2013 academic course we have accomplished all the activities again, obtaining several assessment reports, in particular those for the enhanced game-like simulations and those for the new visualization-based ones [17] which have been used by different groups of students. We are currently analyzing all these reports in order to carry out further improvements.

In this way, by adopting our process model we were able to effectively coordinate the different kinds of participants involved in the production of interactive simulations. Besides, thanks to the continuous underlying assessment process, the development of the different models and tools was driven by the needs of their users: instructors and students. In this way, starting from the kind of exercise typically used to teach the fundamentals of attribute grammars, and directed by the instructors' and the students' assessments, we have developed an educational software system able to generate two kinds of non-trivial interactive simulations.

5. DISCUSSION

Once examined our generative process model, it is possible to compare it with the works introduced in section 2. In particular:

- Concerning the approaches to the development of general-purpose educational systems [3][5][9][12], while these efforts are oriented to develop any type of educational system and software, our proposal is more focused on the

development of a particular type of assets (educational simulations).

- Concerning specific process models for the development of educational games, while according to the process model in [14] instructors must face the complexities of game design by using suitable domain-specific languages, we adopt a more specific and narrowed approach: instructors are focused on the design of exercises that are automatically mapped on working interactive simulations, which can adopt the form of educational serious games. Thus, in our approach instructors are released from having any game design expertise and they are focused on what they are expert: devising quality learning materials. In this sense, our proposal is closer to [10], although it does not compromise with any underlying framework, however. In addition, we make a strong emphasis on the integration of the assessment aspect in the different phases of the process.
- Generative learning objects [6][13] in their turn fit well in our approach, since simulations of a particular kind could well be conceived as instances of the same generative learning object. For this purpose, parameterization of learning objects would be carried out using sophisticated domain models instead of plain collections of attribute-value pairs. Resulting generative learning objects based on rich domain models would be very similar to the works described in [4][8][21].

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a process model for the generative development of interactive simulations, which is specially targeted to Engineering Education settings. This process model encourages the automatic generation of interactive simulations from batteries of exercises, and it integrates continuous assessment as an orthogonal aspect to all the activities of the process. The development of *Evaluators*, a system that use interactive simulations as support to the learning of basic concepts in introductory Compiler Construction courses, has illustrated the feasibility of such a process model. It has also illustrated the worth of the continuous assessment in order to make possible the continuous enhancement of not only the final simulations produced but also of the overall system (included the domain models and the production tools).

As future work we plan to refine our process model by applying it to the production of other educational systems in Computer Science and Engineering Education (e.g., computer programming and application development for mobile devices).

7. ACKNOWLEDGMENTS

Projects and grants TIN2010-21288-C02-01, TIN2009-13692-C03-03 and EDU/3445/2011 partially supported this research. Our thanks to Rafael Fernández-López and Ángel Valero-Picazo for their contribution to the first version of *Evaluators*.

8. REFERENCES

[1] Adams, W.K., Reid, S., LeMaster, R., McKagan, S.B., Perkins, K.K., Dubson, M., and Wieman, C.E. (2008). A Study of Educational Simulations Part I - Engagement and Learning. *Journal of Interactive Learning Research* 19 (3). 397-419.

[2] Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D. (2007). Compilers: principles, techniques and tools (second edition), Addison-Wesley.

[3] Allen, CW. (2006). Overview and Evolution of the ADDIE Training System. *Advances in Developing Human Resources*, 8(4). 430-441.

[4] Castro-Schez, J.J., Redondo, M.A., Gallardo, J. and Jurado, F. (2012). Designing and developing software for educative virtual laboratories with language processing techniques: lessons learned in practical experiments. *Journal of Research and Practice of Information Technology*, 44(3). 289-308

[5] Costa, A.P., Loureiro, M.J., and Reis, L.P. (2009). Development Methodologies for Educational Software: The Practical Case of Courseware SERE. *International Conference on Education and New Learning Technologies (EDULEARN09)*. 5816-5825.

[6] Damaševičius, R. and Štuikys, V. (2008). On the Technological Aspects of Generative Learning Object Development. ISSEP: *3rd International Conference on Informatics in Secondary Schools - Evolution and Perspectives*. 337-348.

[7] Deshpande, A.A. and Huang, S.H. (2011). Simulation games in engineering education: A state-of-the-art review. *Computer Applications in Engineering Education*, 19(3). 399-410.

[8] Diehl, S., Kerren, A. and Weller, T. (2001). Visual Exploration of Generation Algorithms for Finite Automata on the Web. *Lecture Notes in Computer Science: Implementation and Application of Automata*. 327-328.

[9] García-Pacheco, I . and Gracía-Matías, J. (2008). A Methodology Based on Effective Practices to Develop Educational Software. *Computación y* Sistemas, 313-332.

[10] Gómez-Martín, M.A., Gómez-Martín, P.P. and González-Calero, P.A. (2006). Dynamic Binding is the Name of the Game. *International Conference on Entertainment Computing* (ICEC'06). 229-232.

[11] Jiménez-Díaz, G., Gómez-Albarrán, M. and González-Calero, P.A. (2007). Pass the Ball: Game-Based Learning of Software Design. *International. Conference on Entertainment Computing* (ICEC'07). 49-54.

[12] Lage, F.J., Zubenko, Y. and Cataldi, Z. (2001). An Extended Methodology For Educational Software Design: Some Critical Points. *31st Annual Frontiers in Education Conference, 2001*. 13-18.

[13] Morales, R., Leeder D. and Boyle, T. (2005). A Case in the Design of Generative Learning Objects (GLO): Applied Statistical Methods GLOs. ED-MEDIA: *World Conference on Educational Multimedia, Hypermedia and Telecommunications*. Vol. 2005, No. 1. 2091-2097.

[14] Moreno-Ger, P., Martínez-Ortiz, I., Sierra, J.L. and Fernández-Manjón, B. (2008). A Content-Centric Development Process Model. IEEE *Computer*, 41(3). 24-30.

[15] Paakki, J. (1995). Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computer Surveys*, 27(2), 196-255.

[16] Rodríguez-Cerezo, D., Gómez-Albarrán, M. and Sierra, J.L. (2011). From Collection of Exercises to Educational Games: A Process Model and a Case Study. *11th IEEE International Conference on Advances Learning Technologies* (ICALT'11). 282-284.

[17] Rodríguez-Cerezo, D., Gómez-Albarrán, M. and Sierra, J.L. (2013). Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts. *18th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE 2013). 28-33.

[18] Rodríguez-Cerezo, D., Sarasa, A., Gómez-Albarrán, M., and Sierra, J.L. (2012). Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game Based Approach. *International Simposium on Computers in Education (SIIE'12)*. 1-6.

[19] Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M. and Sierra, J.L. (2013). Serious games in tertiary education: A case study concerning the comprehension of basic concepts in computer language implementation courses. *Computers in Human Behavior* (in press). Online access: http://dx.doi.org/10.1016/j.chb.2013.06.009

[20] Sadler, D.R. (1989). Formative Assessment and the Development of Instructional Systems. *Instructional Science*, 18(12), 119-144

[21] Sierra, J.L., Fernández-Pampillón, A.M. and Fernández Valmayor, A. (2008). A. An environment for supporting active learning in courses on language processing. *13th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE '08). 128-132.

6.4 From collections of exercises to educational games: A process model and a case study

Cita completa:

Rodriguez-Cerezo, D., Gomez-Albarrán, M., Sierra, J. L. From Collections of Exercises to Educational Games: A Process Model and a Case Study. En ICALT'11: 11th IEEE International Conference on Advanced Learning Technologies, 11: 282-284. 2011.

Resumen original de la contribución:

In this paper we present a process model for the construction of educational games, which is driven by the exercises proposed by the instructors. This model promotes a tight collaboration between instructors and game developers during game construction. It also incorporates formative evaluation accomplished by instructors and students. In order to illustrate the process, we use an experience concerning game-based learning in a Compiler Construction Course.

Referencia de citas bibliográficas:

[60][75][109][120][158][164]

2011 11th IEEE International Conference on Advanced Learning Technologies

From Collections of Exercises to Educational Games: A Process Model and a Case Study

Daniel Rodríguez-Cerezo, Mercedes Gómez-Albarrán, José-Luis Sierra
Dep. Ingeniería del Software e Inteligencia Artificial. Facultad de Informática.
Universidad Complutense de Madrid. 28040 Madrid (Spain)
drodricerezo@gmail.com, albarran@sip.ucm.es, jlsierra@fdi.ucm.es

Abstract— **In this paper we present a process model for the construction of educational games, which is driven by the exercises proposed by the instructors. This model promotes a tight collaboration between instructors and game developers during game construction. It also incorporates formative evaluation accomplished by instructors and students. In order to illustrate the process, we use an experience concerning game-based learning in a Compiler Construction Course.**

Keywords: ***educational games, development process model, Computer Science education, attribute grammars***

I. INTRODUCTION

Building an educational game is not an easy task and requires the effective collaboration of three different types of actors: (a) instructors, who know how to design learning contents and strategies, (b) game designers and other IT professionals, who master the technical skills and knowledge needed to build sophisticated games; and, (c) students that help to assess the final games.

In this paper we propose a process model to build educational games that supports this collaboration. This process model uses the exercises proposed by the instructors as a keystone element on which to base the overall life-cycle. Indeed, this structure is used (i) to provide suitable authoring tools for the exercise descriptions and solutions, and (ii) to propose adequate metaphors that reformulate the process of solving exercises as the process of playing the resulting game. This process model is based on our previous work on the production of learning contents and tools [6], as well as in the development of educational simulations and games [2].

II. THE EDUCATIONAL GAME EXERCISE-DRIVEN PRODUCTION PROCESS MODEL

The products and activities involved in our exercise-driven production process model are sketched in Figure 1. The main actors in each activity are also depicted.

The objective of the *conception* activity is to provide the *domain models* that will drive the rest of the process. On the one hand, it is necessary to provide a model that characterizes, both, the exercise descriptions and the expected solutions: the *exercise model*. On the other hand, it is necessary to model educational games based on the mental processes required to solve this kind of exercises. The *game model* gathers this information. This model will be largely based on a suitable metaphor allowing the transformation of collections of exercises (i.e., instances of the *exercise model*) into running games.

The *tool provision* activity is oriented to provide the toolset required to successfully implement the game-based learning system: the *authoring tool*, the *game generator* and the *analysis tool*. The *tool provision* activity uses the domain models to create these tools. The continuous iteration of the *conception* and the *tool provision* activities constitutes the *tool provision loop*.

Instructors use the *authoring tool* in the *exercise authoring* activity to create collections of *exercises*. In its turn, the *game generator* will be able to automatically generate the *games* from the coded exercises. The foundations of this generation process are in the aforementioned metaphor of the *game model*. The generation itself is carried out in the *game generation* activity. The continuous interplay of *authoring* and *game generation* activities yield a *game provision loop*.

Finally, the *analysis tool* facilitates instructors to study the students' behavior while playing the games during the *playing* activity. As result of this activity, the games log students' behavior. These *logs* can be analyzed with the *analysis tool* during the *student assessment* activity in order to assess the progress of the students. The students' use of the game and the instructors' assessment of student progress constitute the *learning loop*.

It is remarkable how a formative assessment, oriented to improve the learning system, can be smoothly incorporated into the different activities of the process model. Indeed, instructors can assess different aspects of the *authoring tool* during the *exercise authoring* activity (e.g., is the tool easy to use? does the tool accomplish all the expressivity required to encode the exercises?). In their turn, students can evaluate different aspects of the games while *playing* them (e.g., usability; degree of entertainment; subjective pedagogical utility, etc). Finally, instructors can assess the suitability of the analysis tool and the pedagogical efficacy of the game in the *student assessment* activity. These assessments can reactivate upper loops from lower ones.

III. AN EXAMPLE: PRODUCTION OF EDUCATIONAL GAMES IN A COMPILER CONSTRUCTION COURSE

We have applied our process model to develop *Evaluators*, a game-based learning system oriented to teach fundamental concepts of computer language implementation. The experience was carried out in a Compiler Construction course at UCM (Complutense

978-0-7695-4346-8/11 $26.00 © 2011 IEEE
DOI 10.1109/ICALT.2011.88

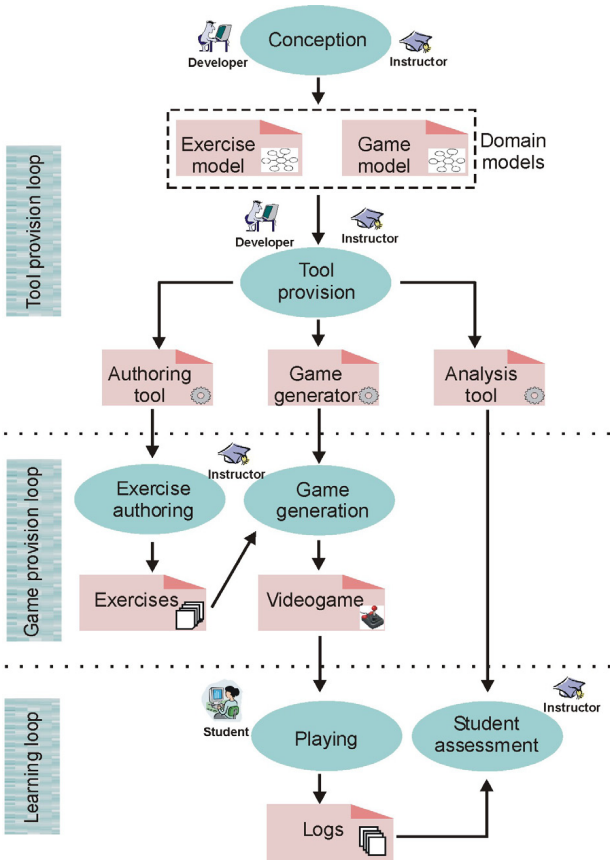


Figure 1. Products, activities and participants

University of Madrid, Spain). *Evaluators* helps to understand basic concepts of *attribute grammars* (AGs) [4], a formalism widely used to describe the syntax and semantics of computer languages. *Evaluators* is similar in spirit to those systems described in [1][2]. However, it differs in the learning domain, and we have rigorously adopted the process model described in this paper to develop it.

A. Conception

AGs are artifacts used to specify how to process the sentences of a language. The computational model of AGs builds a parse tree that represents the structure of each sentence. Semantic attributes are associated with the nodes of this tree and hold values during the processing of sentences. Besides, a set of semantic equations describe how to compute the values of the semantic attributes.

AGs do not impose any particular order to compute the values of the parse tree attributes. The only constraint that should be satisfied is to have previously computed the values of all the other attributes required by the semantic equations. Therefore, it is very important for students in Compiler Construction courses to understand this dependency-driven computation style. For this purpose, instructors at UCM use exercises whose description proposes a simple language, a processing problem, a formalization based on an AG, and a sentence in the language. The exercise solution consists of determining a possible order in which the attributes in the parse tree of the sentence can be evaluated, as well as the resulting values of these attributes.

Evaluators is based on this type of exercises. Its exercise model is adjusted to the specification described above and lets us describe the attributed parse tree, the attribute values and the dependencies between the attributes. The dependencies are, in this case, an implicit characterization of all the valid solutions for an exercise.

The game model, in its turn, is based on a metaphor that adopts the following conventions:

- Syntax trees are viewed as maze-like worlds where rooms represent tree nodes and arcs are represented as corridors. Each room has a table with boxes on it. These boxes represent instances of semantic attributes and can contain objects (the values of the attributes).
- Each game includes an avatar which can be commanded by the student. The student uses the avatar to carry objects from box to box using an *inventory*, thus reproducing the attribute evaluation process. The action of activating a box is a metaphoric way of expressing that an attribute has synthesized its value.

The main objective in the game is to move the contents of the appropriate boxes through the maze and combine them to activate all the boxes.

B. Tool Provision, Exercise Authoring and Game Generation

Evaluators authoring tool lets the instructors edit the attributed syntax tree, by defining the attributes in each node and the dependencies between these attributes (Figure 2a).

The game generator lets customize games by choosing sequences of exercises, in such a way that each exercise is viewed as a different level in the game (Figure 2b). Once the collection of exercises is selected by the instructor, the generation process proceeds automatically, yielding a running game as output (Figure 2c).

The analysis tool makes it possible to examine the students' behavior. For this purpose, generated games log those actions of students that correspond with solution steps of the problem posed. Then the analysis tool shows to the instructor graphically the solution strategy followed by the student (Figure 2d).

Regarding formative assessment, we accomplished a little survey among the instructors that used the authoring and analysis tools in the first term of the 2010 edition of the Compiler Construction course. The game generator got very positive comments. However, the authoring tool was criticized by the educators because the creation of complex syntax trees and their dependencies are tedious. In order to solve this problem, we are working on a tool that will be able to automatically construct the trees from the

exercises' descriptions given in a specific language, in a similar way as those proposals described in [3][5] do.

C. *Playing and Student Assessment*

Our initial experience using *Evaluators* has been positive. The system was used by the students enrolled in the Computer Construction course during the first term of the 2010 edition, and instructors observed a positive attitude to the proposed game-based learning strategy. Although globally the students' progress was similar to the one observed in previous editions, the difference was in a massive participation of the students enrolled in the course. Indeed, it makes apparent the motivating factor of this kind of approaches with respect to more conventional alternatives (e.g., solving the exercises with paper and pencil).

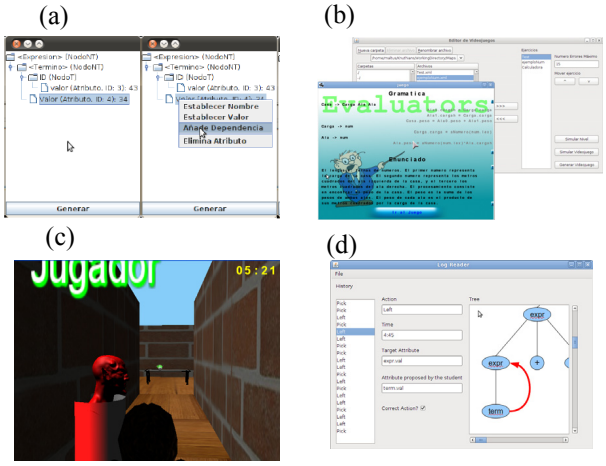


Figure 2. Screenshots of the authoring and the analysis tools, the game generator and one of the generated games.

We also conducted a survey among students to formatively assess *Evaluators*. Survey results were promising. *Evaluators* obtained success and popularity within this group of students as a complementary tool for Compiler Construction lectures. Many students considered that it is enjoyable and useful for understanding the different aspects related to AGs (i.e., semantic attributes, semantic equations, processing). It also allowed us to be aware of some aspects of the game that need to be improved. The game does not show a very steep learning curve, but some difficulties of use remain (difficulties in the commands and a not-friendly interface). We are now undertaking an improvement of the game to solve these shortcomings.

Finally, regarding instructors' assessment of the analysis tool, it was very well evaluated in the survey mentioned in the previous subsection. As a minor matter, some educators pointed to include more actions like reporting inactivity of the user during the play or automating the detection of unnecessary behaviors.

IV. Conclusions and Future Work

In this paper we have presented a process model for the production of high-quality and pedagogically focused educational games. This model is largely guided by the types of exercises used by the instructors in their learning domains, and it strongly encourages the active participation of instructors. Besides, the process model incorporates an important *meta-level* layer, in which instructors collaborate with developers to provide and enhance different supporting tools. In this process model, the formative evaluation accomplished by instructors and students is essential to detect and make up for deficiencies that should be solved in order to finally obtain successful learning processes.

We have also illustrated the approach with *Evaluators*, which is focused on teaching the fundamental concepts of syntax-directed translation formalized with AGs. In *Evaluators* development we have used an appropriate metaphor to map the concepts of the formalism into games, and we have followed the rest of the process, including a careful formative assessment of the overall learning system.

We are currently working on improving the *Evaluators* toolset. In the short term, we hope to expose *Evaluators* to a more thoroughly evaluation process during next editions of the Compiler Construction course, and also with instructors outside the UCM.

We are also planning to experiment with other domains: literary analysis, classification of corpus of historical documents, formal logic, leadership in education, and computer programming in CS.

ACKNOWLEDGEMENTS

This research was partially supported by project grants TIN2010-21288-C02-01, TIN2009-13692-C03-03, and TIN2007-68125-C02-01. We also would like to thanks Rafael Fernández-López and Ángel Valero-Picazo by their invaluable contribution to the implementation of a previous version of *Evaluators*.

REFERENCES

[1] Gómez-Martín, M-A., Gómez-Martín, P-P., González-Calero, P-A. 2005. Game-Driven Intelligent Tutoring Systems. Int. Conference on Entertainment Computing ICEC'04.

[2] Jiménez-Díaz, G., Gómez-Albarrán, M., González-Calero, P-A. 2007. Pass the Ball: Game-Based Learning of Software Design. Int. Conference on Entertainment Computing ICEC'07

[3] Mernik, M., Žumer, V. 2003. An Educational Tool for Teaching Compiler Construction. IEEE Trans. on Education, 46 (1), 61–68.

[4] Paakki, J. 1995. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computer Surveys, 27, 2, 196-255.

[5] Sierra, J.L., Fernández-Pampillón, A.M., Fernández-Valmayor. 2008. A. An environment for supporting active learning in courses on language processing. 13th annual conference on Innovation and technology in computer science education, ITiCSE '08

[6] Temprado-Battad, B., Sarasa, A., Sierra, J.L. 2010. Using Attribute Grammars to Manage the Production and Evolution of e-Learning Tools. 9th International Conference on Advanced Learning Technologies ICALT'10.

6.5 Interactive educational simulations for promoting the comprehension of basic compiler construction concepts

Cita completa:

Rodriguez-Cerezo, D., Gómez-Albarrán, M., Sierra-Rodríguez, J. L. Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts. En ITiCSE'13: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, 28-33. 2013.

Resumen original de la contribución:

Evaluators 2.0 is an educational software system that lets instructors in introductory compiler construction courses generate interactive simulators from batteries of exercises concerning basic concepts in attribute grammars. The system also makes it possible to analyze the activities of the students who use these simulators. Students interact with the simulators generated to decide the evaluation order of semantic attributes in the attributed syntax trees, and they receive immediate feedback about the actions performed. Thus, these simulations help students to understand the fundamental concepts of the formalism of attribute grammars and of its underlying computational model. This paper describes the different software tools that constitute Evaluators 2.0 as well as the nature of the simulators generated, and also reports on different assessments of the system involving both instructors and students.

Referencia de citas bibliográficas:

[4][5][10][20][32][40][50][54][65][88][89][109][120][127][137][158][165][176][179]

Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts

Daniel Rodríguez-Cerezo
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases s/n
28040 Madrid (Spain)
+34913947606
drcerezo@fdi.ucm.es

Mercedes Gómez-Albarrán
Fac. Informática
Universidad Complutense de Madrid
Ciudad Universitaria 28040 Madrid (Spain)
+34913947561
albarran@sip.ucm.es

José-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases s/n
28040 Madrid (Spain)
+34913947548
jlsierra@fdi.ucm.es

ABSTRACT
Evaluators 2.0 is an educational software system that lets instructors in introductory compiler construction courses generate interactive simulators from batteries of exercises concerning basic concepts in attribute grammars. The system also makes it possible to analyze the activities of the students who use these simulators. Students interact with the simulators generated to decide the evaluation order of semantic attributes in the attributed syntax trees, and they receive immediate feedback about the actions performed. Thus, these simulations help students to understand the fundamental concepts of the formalism of attribute grammars and of its underlying computational model. This paper describes the different software tools that constitute *Evaluators* 2.0 as well as the nature of the simulators generated, and also reports on different assessments of the system involving both instructors and students.

Categories and Subject Descriptors
K.3 [**Computers and Education**]: (K.3.1) Computer Issues in Education – *Computer-assisted Instruction (CAI)*; (K.3.2) Computer and Information Science Education – *Computer science* education. D.3 [**Programming** Languages]: (D.3.4) Processors – *Translator writing systems and compiler* generators.

General Terms
Human Factors, Languages.

Keywords
Education in Compiler Construction, Authoring Tool, Analysis Tool, Interactive Educational Simulation, Attribute Grammar

1. INTRODUCTION
Computer language implementation has proven to be a key subject to understand new programming approaches based on model-driven development and domain-specific languages [7][10]. However, students usually find this subject a very hard one due to its abstract nature and the large number of concepts and techniques that it involves [1]. This fact can be a cause of demotivation, and, therefore, of poor performance by students.

ITiCSE'13, July 1–3, 2013, Canterbury, England, UK.

Based on our experience teaching this subject at the Complutense University of Madrid (Spain), we advocate a clear separation of the specification and the implementation aspects of language processors. Thus, we have adopted *syntax-directed translation* [2] as a central paradigm for architecting language processors and *attribute grammars* [13] as a basic model for the specification of the processing tasks. We realize that the success of this teaching strategy depends considerably on whether students understand the formalism of attribute grammars at the early stages of the course. Indeed, if students are unable to correctly assimilate those basic concepts, they become lost when applying the formalism to the modeling of different computer language processing problems. As a result, they fall into the aforementioned demotivation state.

In order to alleviate this problem, we started by making a special emphasis on these basic concepts of the formalism and, in particular, the concepts concerning the operational semantics of attribute grammars. We created batteries of exercises focused on such basic concepts. Each exercise consisted of: (i) an informal description of a language processing problem, (ii) a formalization of this processing problem by means of an attribute grammar, (iii) a sentence in the processed language, and (iv) the attributed parse tree associated with this sentence. To solve the exercise the student had to explain how to evaluate the values of the semantic attributes in the parse tree.

This first attempt had a positive effect on students' assimilation of the formalism but showed a kind of rote learning: many students tended to memorize the patterns of the solutions in our sample exercises and to apply these patterns in an indiscriminate way, without a real understanding of what was going on. To prevent this shortcoming, we decided to develop *Evaluators*, a system that generates serious educational games from batteries of exercises [15].

Games generated in *Evaluators* involve students in interactive simulations of semantic evaluation processes, their role being to determine the correct evaluation order for semantic attributes. For this purpose, attributed parse trees were mapped onto mazes with rooms (one room per node in the parse tree). Attributes decorating each node were mapped, in turn, in boxes placed on tables located in the rooms. Thus, in order to determine the right evaluation order, students must move objects from box to box while obeying the constraints imposed by the formal specification (see Figure 1). These serious games provided immediate feedback on student actions and registered these actions for later analysis by the instructors.

Evaluators was welcomed both by students and instructors, who rated it to be a useful instrument to teach and learn attribute grammars (see [15]). However, they reported an undesirable

effect: the game component, although rewarding for the students, substantially increased the amount of time required to solve the exercises. In order to avoid this pitfall and, at the same time, encouraged by the impact of visualization in education [14][18] and its suitability to the computational model of attribute grammars, we have redesigned the game-based system to generate more straightforward visual simulators from the exercises. These simulators preserve the positive properties of the serious games generated by *Evaluators*, while avoiding the penalty in time required to solve the exercises. The result is a system called *Evaluators 2.0*.

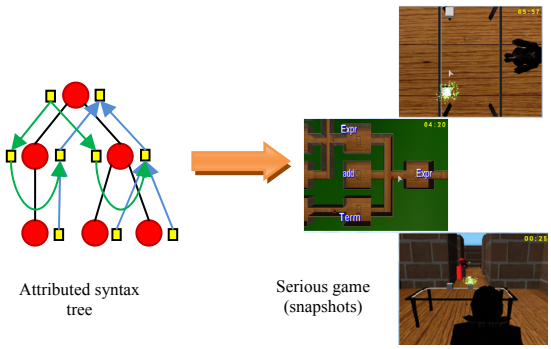


Figure 1. Mapping attributed syntax trees to serious games in *Evaluators*

Evaluators 2.0 is similar to other educational software oriented to simulate different aspects of computer language processors with an education purpose (e.g, proposals centered on the simulation of theoretical machines [9][19], or proposals focused on the simulation of computer language-related algorithms and the visualization of the involved internal structures [5][8][17]). The main difference is that *Evaluators* 2.0 is focused on the declarative formal specification of syntax-directed processes using attribute grammars, instead of the underlying theoretical machines (like [9][19]) or the implementation details of algorithms and data structures of final processors (and meta-processors) (like [5][8][17]). Also, *Evaluators* 2.0 is related to proposals based on high-level specifications for the teaching and learning of language processors (e.g., [12][16]). However, proposals like the one described in [12] use production meta-tools instead of tools specifically oriented to educational purposes. Still, tools like the one presented in [16], which were designed with an eminently educational mission, are usually focused on more advanced stages of the learning process (i.e., to construct formal specifications based on attribute grammars instead of helping students to comprehend them).

The rest of the paper describes *Evaluators* 2.0, and provides preliminary data on its assessment with instructors and students. Indeed, Section 2 describes the system itself, focusing on its different tools and the nature of the interactive simulations generated. Section 3 presents the assessment results. Finally, Section 4 presents some conclusions and lines of future work.

2. *EVALUATORS* 2.0

In order to understand the essence of Evaluators 2.0 we will first describe the attribute grammar formalism. Then we briefly describe the different tools that make up the system. Finally, we present the nature of the simulations developed with *Evaluators* 2.0.

2.1 Learning Domain: Attribute Grammars

Attribute grammars were developed by D.E. Knuth [11] as an extension of context-free grammars in order to describe syntax-directed language processing tasks. Figure 2a shows an example of attribute grammar that formalizes the evaluation of simple arithmetic expressions.

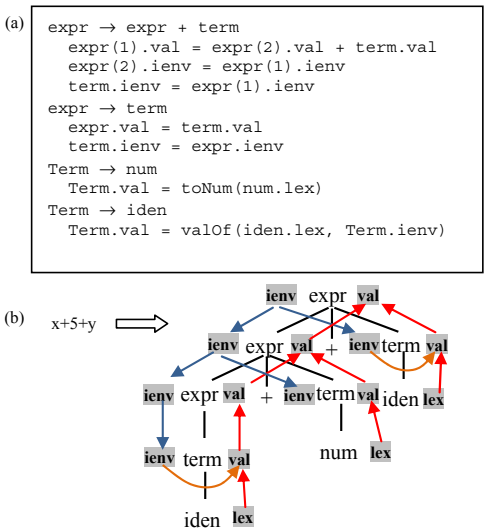


Figure 2: (a) An example of attribute grammar; (b) Attributed parse tree associated with a sentence, along with the attribute dependency graph.

Thus, attribute grammars are built on context-free grammars. *Context-free grammars* use *terminal symbols* to represent the basic elements that make up the language, *non-terminal symbols* to denote composite structures, and *syntax rules* to specify how non-terminal symbols can be structured as sequences of other non-terminal and terminal symbols. For instance, in Figure 2a, `num`, `iden` and `+` are examples of terminal symbols (these symbols represent, respectively, numbers, identifiers and the *addition* operator), `expr` and `term` are examples of non-terminal symbols (they represent, respectively, additive and basic arithmetic expressions), and `expr → expr + term` is an example of a syntax rule (it specifies that an expression may be constituted by another expression –the first operand – followed by +– the operator – and by a term –the second operand).

Context-free grammars impose tree–like structures known as *parse trees* on sentences of the modeled computer language. In the parse tree for a sentence, inner nodes are labeled by non-terminal symbols, their children are obtained by applying syntax rules, and leaves are labeled by terminal symbols (or by λ, the empty string). In addition, the sentence is retrieved by reading the leaves from left to right (while ignoring λ). However, context-free grammars do not go beyond assigning these tree-shaped structures. Therefore, in order to describe processing, other means need to be used. Fortunately, a natural way of specifying how the sentences of a language can be processed is to use the parse trees to drive such a process; this is known as *syntax-directed translation*. In order to specify syntax-directed translation tasks, Knuth extended context-free grammars with the following artifacts:

- A set of *semantic attributes* associated with the symbols of the grammar. These attributes can be conceived as placeholders in which to store the values obtained during the

processing of the sentences. There are two kinds of attributes: *synthetized* and *inherited* attributes. Synthetized attributes maintain the results of processing a symbol, and inherited attributes contain contextual information required for this processing.

- A set of *equations* associated with the syntax rules, which specify how to compute the values of the synthetized attributes on the rules' left-hand sides, as well as the inherited attributes on the rules' right-hand sides.

Thus, the computational model of attribute grammars is based on (i) building a syntax tree for the processed sentence, (ii) associating semantic attributes to each node on the tree in order to yield an *attributed syntax tree*, and (iii) carrying out a *semantic evaluation process* to determine, based on the semantic equations, the values of those attributes. For instance, Figure 2b shows a sentence, along with its attributed syntax tree, and also a *dependency graph* showing the dependencies imposed on attributes by semantic equations. Indeed, a key aspect of the formalism is that attribute grammars do not impose a particular evaluation order on attributes. The only constraints imposed on the evaluation order are given by the dependency graphs: before computing the value of an attribute it is necessary to compute the values of all the attributes on which it depends. Therefore, it is very important for students to internalize this computation model as a necessary step in further developing their own specifications based on attribute grammars. It is indeed the main aspect addressed by *Evaluators* 2.0.

2.2 An Overview of *Evaluators* 2.0

Evaluators 2.0 integrates different tools: an authoring tool for creating language processing exercises, a customizer tool for generating semantic evaluation simulators for these exercises, and an analysis tool for evaluating student performance during the use of these simulations. The *Evaluators* 2.0 workflow is similar to the one followed by *Evaluators* (see [15]). However, the software generated from the exercises is not a game-like simulator, as in *Evaluators*, but is based on more direct simulations of the semantic evaluation process. Figure 3 summarizes the use of *Evaluators* 2.0 by the different kinds of actors implied (*instructors* and *students*). According to this workflow:

- First, instructors use the authoring tool to create batteries of exercises with the structure mentioned in Section 1. The authoring tool lets instructors define language processing tasks by providing their informal and formal descriptions, the latter in the form of attribute grammars. For this purpose, the tool provides an appropriate editor, and it supports a domain specific language for describing attribute grammars by using a notation similar to the notation used in compiler construction lectures and courses. Once the language processing tasks are defined, instructors can select these descriptions and specify concrete sentences in order to define concrete exercises. Also, the authoring tool lets instructors select, from all the attributes involved in each exercise, those that have already been calculated. This feature lets instructors focus students' attention on certain semantic aspects (e.g., translation of control structures) without worrying about simpler, already mastered aspects (e.g., type-checking or evaluation of expressions). Finally, instructors can define the maximum number of mistakes that students can make while solving the exercise. It is worthwhile to note that instructors do not need to provide the attributed parse tree explicitly, nor the value of the attributes or the possible constraints that apply during the solution of the problem (i.e, the dependencies among the attributes). On the contrary, the authoring tool automatically derives this information from the formal specification based on attribute grammars and the sentence. This remarkable functionality of the authoring tool substantially facilitates the development of the exercises.

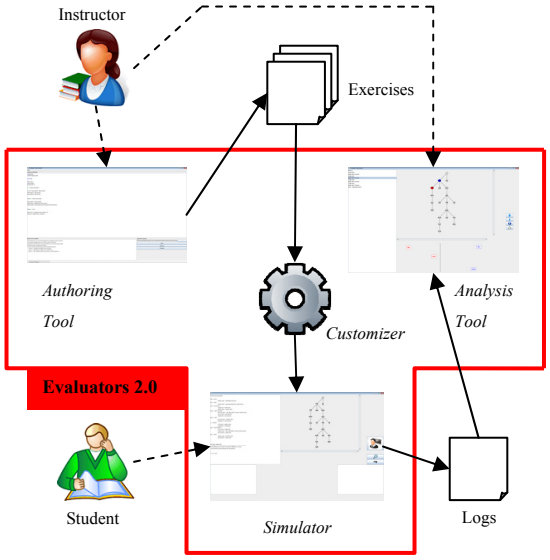


Figure 3 *Evaluators* 2.0 workflow.

- When the instructors finish the definition of the exercises, they can use a customizer tool to generate the simulator. For this purpose, they select a suitable set of exercises, each one representing a simulation level that must be achieved by the students. Once complete, the customizer automatically generates the simulator.
- Once the simulator is available, students can interact with it in order to solve the exercises. The simulator will provide a visualization of the syntax tree for each exercise, on which the students have to perform the semantic evaluation process. Making use of the attribute dependencies and the other information associated with the correct solutions, the simulator can provide students with appropriate feedback concerning correct/incorrect actions, attribute values, etc. Besides, it registers student activity in log files.
- Finally, instructors can analyze student behavior during the interactive simulation by using the analysis tool. For this purpose, the tool uses the logs stored by the simulator and provides a visual environment able to display the actions registered in the logs. Thus, instructors can detect recurrent mistakes and elaborate pertinent reinforcement actions.

2.3 The Simulators

As indicated before, *Evaluators* 2.0 generates interactive educational simulators that offer visual representations of attributed syntax trees (Figure 4a) and students interact with them to solve the associated exercises. Once an exercise is completed, the simulator switches to the next simulation level corresponding to the next exercise defined by the instructor. In more detail, the information and interaction offered by the simulator to the users for each level is as follows:

- Along with the syntax tree visualization, the simulator presents the exercise description that contains the informal

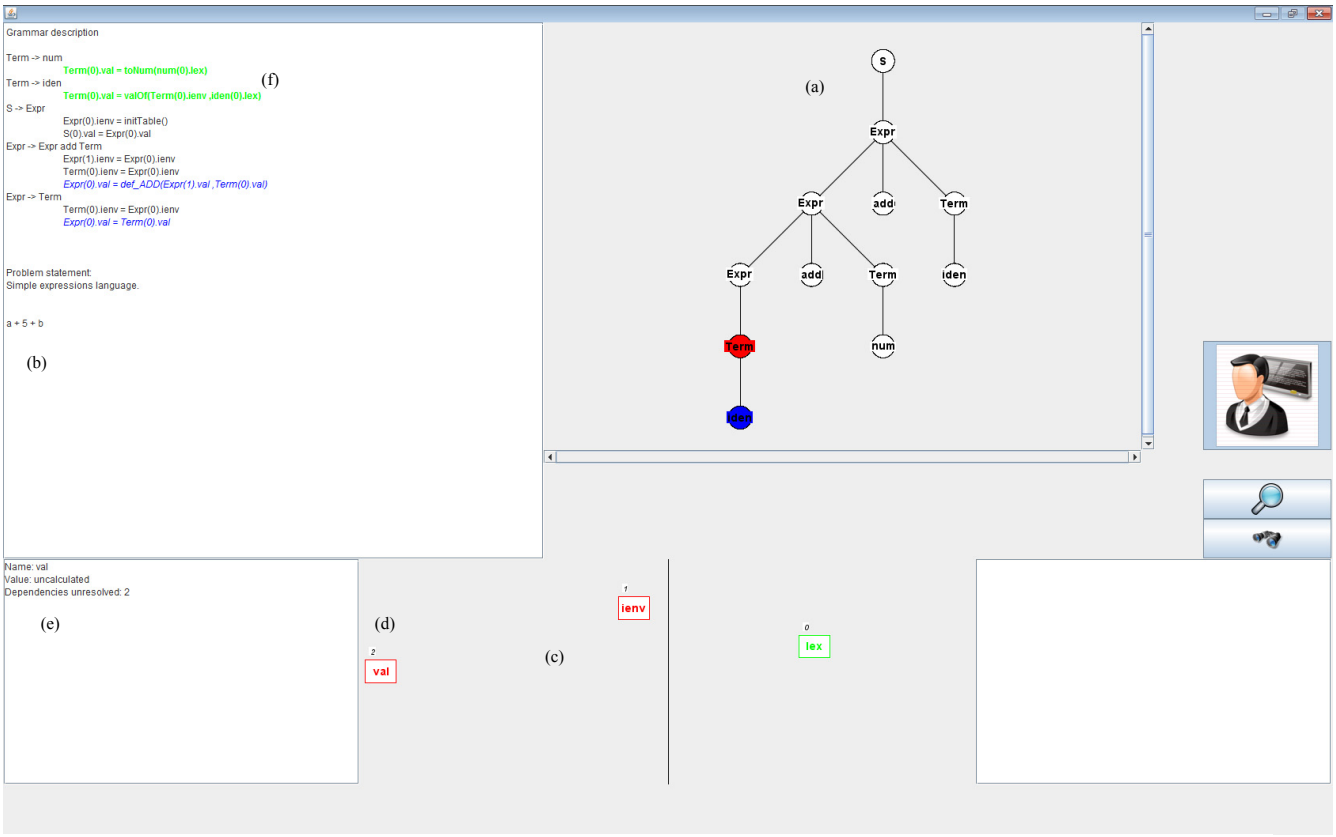


Figure 4 Screenshot of a simulator generated by *Evaluators* 2.0 (see main text for further explanations)

description of the language processing task, as well as its formal description (i.e., the attribute grammar) (Figure 4b).

- Then, users can explore the nodes of the syntax tree in order to visualize the attribute instances associated with them (Figure 4c). The tool makes it possible to simultaneously explore pairs of nodes among which the student can propagate the values of semantic attributes.
- The attributes' instances are represented as boxes with the attribute name and the number of still unresolved dependencies attached (Figure 4d).
- Besides, students can consult each box to obtain more details about the attribute instances (Figure 4e), such as their value (whenever it is ready), their name, etc.
- Also, if the student activates the additional support provided by the simulator, when querying an attribute the semantic equations in which it intervenes will be highlighted in the attribute grammar (see Figure 4f). For this purpose, the highlighting color depends on whether the attribute instance is used to calculate the value of another attribute instance, or whether the equation is actually used to compute the value of this attribute.
- Finally, in order to involve students in the simulation of the semantic evaluation process, the simulator generated lets students drag a box (representing an attribute used in a computation) and drop it into another box (the attribute which is being computed). For this purpose, the simulator checks the basic constraints imposed by the attribute grammars: (i) the value of the attribute dropped should have been previously calculated, and (ii) the value of the attribute dropped is necessary, according to the corresponding semantic equation, to calculate the value of the attribute which is being computed. Thus, the simulator will provide appropriate feedback after this dropping action, indicating the success or failure of the action. Besides, if the user fails, the feedback will inform about the cause of the mistake.

During the interactive simulation process described, when students go beyond the maximum number of mistakes allowed in a level (defined by the instructors in the corresponding exercise), the level will be restarted.

3. ASSESSMENT

In order to assess the utility of *Evaluators* 2.0 we ran three different evaluation efforts:

- An informal survey directed at instructors in Computer Science at the Complutense University, including both lecturers in compiler construction and lecturers in other topics. The goal of this survey was to research on the perceived utility of *Evaluators* 2.0 as a tool for generating interactive educational simulations, as well as the teachers' perception about the educational utility of the simulations generated.
- A structured interview targeted at students of an introductory course on Compiler Construction at the Complutense University. The goal of this interview was to investigate the utility of the interactive simulations perceived by the students, as well as other aspects concerning user satisfaction, motivation and usability.
- A controlled experiment involving two groups of volunteer students from the aforementioned Compiler Construction course. The goal of this experiment was to assess the educational efficacy of the interactive simulators compared to solving the exercises with paper and pencil.

Next sections summarize the results of these evaluations.

3.1 Evaluation by instructors

In order to investigate the utility of *Evaluators* 2.0 perceived by instructors we involved 10 instructors of Computer Science, whose expertise ranged from language processing to introductory programming. We explained to the instructors the fundamentals of the learning domain (for those instructors not familiar with attribute grammars), as well as the main ideas behind *Evaluators* 2.0. Then we showed instructors a short demo of the tool, give them the choice to *play* with the system, and, finally, asked them for their opinion. As a result, we realized that the tools and the simulations generated were positively rated by the instructors interviewed. Indeed:

- In general, the authoring tool was considered to be an intuitive tool for defining the kind of exercises described above, although some instructors not related to the field of computer language processing found some aspects rather confusing and stated the need for better online help. However, the instructors familiar with Compiler Construction were delighted by the easiness of use, and, in particular, by the adequacy of the attribute grammar description language.
- Concerning the customizer tool, it is remarkable that all the instructors were delighted by the exercise-driven generative approach to the construction of interactive educational simulations adopted in *Evaluators* 2.0. All of them highlighted it as a cost-effective approach to the incorporation of this kind of technology in the educational process, and many of them manifested the usefulness of similar tools for their respective learning domains.
- Likewise, generated simulations were considered a natural way of experimenting with the semantic evaluation process of attribute grammars. Some teachers, again more frequently not familiar with the topic, found the mechanism of selecting pairs of nodes a bit tricky and confusing, and, in particular the possibility of selecting non-adjacent nodes, since attribute grammars only permit the propagation of attribute values between adjacent ones. However, it is significant that an instructor with in-depth expertise in the development of educational software pointed out that possibility as beneficial for students' learning.
- Finally, the analysis tool was found to be useful in assessing the students' performance at a glance. Some instructors suggested the incorporation of some support for (semi)automatic assessment using patterns to detect correct and incorrect behaviors, as well as the possibility of applying these patterns massively to evaluate the performance of a group of students as a whole.

3.2 Evaluation by students

The evaluation by students was focused on the simulators generated by *Evaluators* 2.0. For this purpose, we called for volunteers in the current 2012-2013 edition of the aforementioned introductory Compiler Construction course (participation was rewarded as extra credit). 32 students accepted our invitation to participate. From these volunteers, 14 used the generated simulations, after which they were requested to fill out a small survey.

The survey consisted of seven four-point Likert-like scaled questions (the scale employed was *Disagree, Partially Disagree, Partially Agree, Agree*). The questions were adapted from the TUP model for evaluating educational software [4], and are listed in the first column of Table 1. The percentage shown in Table 1 corresponds to the *Agree* and *Partly Agree* answers registered among the students. The students' answers show that:

- Students mostly agree that simulators can help them to comprehend key aspects of attribute grammars: the processing model, semantic equations, and, to a lesser extent, the role of semantic attributes (perhaps due to the focus of the tool on the semantic evaluation process as a whole, the values of the attributes playing a secondary role). They also agreed on the role of the simulators as a complement to classroom sessions, and on the role of the simulations in encouraging furthering studies on the topic (perhaps to a lesser extent, due to the controversial nature of the question).
- Concerning the categorization of attributes as lexical, synthesized and inherited, there was a divergence of opinions, perhaps due to the non-explicit incorporation of this distinction in the tool.
- Finally, it is worthwhile to notice that a significant percentage of the students surveyed found it difficult to notice that they were making mistakes.

Table 1 Student agreement with survey items.

Survey questions	***% agree***
The simulator has helped me better understand the processing model of attribute grammars.	78,57
The simulator has helped me better understand the role of semantic attributes during processing.	64,28
The simulator has helped me better understand what semantic equations do during processing.	85,71
The simulator has helped me better understand the differences between different types of attributes (lexical, synthesized, inherited).	50,00
When I made a mistake it was not easy for me to realize that I was wrong.	42,85
The simulator conveniently complements the explanations given in lectures.	85,71
The simulator has encouraged my motivation to study the processing model of attribute grammars in general.	64,28

3.3 Educational Effectiveness

In order to evaluate the short-term educational effectiveness of the *Evaluators* 2.0 interactive simulations we devised a battery of representative exercises, generated a simulation based on these exercises, and split the 32 aforementioned volunteers into two groups: a *control* group with 18 students, who solved the exercises with paper and pencil, and the aforementioned *experimental* group with 14 students who solved the exercises with the simulator. The division into groups was performed while ensuring homogeneity, according to the results of a pre-test. Next, in a post-test and using paper and pencil, all the students solved an exercise substantially more difficult than those used in the experience and whose solution required a complex multi-pass evaluation strategy.

After analyzing the solutions provided to the post-test exercise, we categorized the students in two classes (*fail* and *pass*), according to whether they gave a correct or incorrect solution to the problem posed, respectively. Results are summarized in Figure 5, which displays the number of students from each group –control and experimental– who passed and failed the post-test. At first glance, we see a noticeable difference between the ratio of students who passed the test in each group, with a clear positive outperformance of the group who used the interactive simulations (odds ratio of 8.85 with respect to this method and the chance to

pass the test). In addition, the dependency between the group –control or experimental– to which the student belongs and student performance was clearly proven significant from a statistical point of view (χ^2 test: p-value = 0.017).

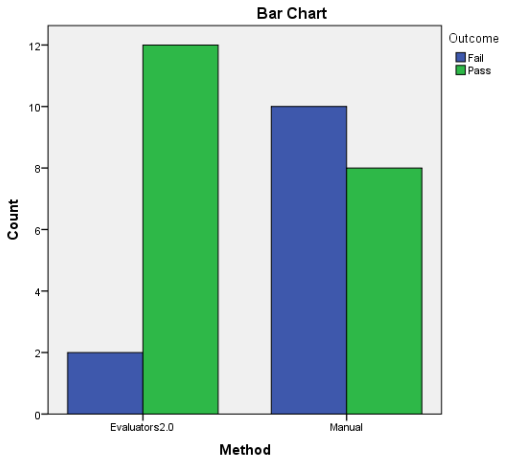


Figure 5 Comparative bar chart of student performance.

4. CONCLUSIONS AND FUTURE WORK

As acknowledged by instructors, *Evaluators* 2.0 is a feasible and cost-effective approach to the generation of interactive educational simulations to help students with the critical aspect of assimilating basic concepts of attribute grammar formalisms. The system improves on a previous, game-based, version (*Evaluators*) while preserving its advantages and improving the aspect concerning the time devoted to solving the exercises. In addition, *Evaluators* 2.0 interactive simulations were welcomed by students, who perceived interactive simulations as a useful way of learning the basic concepts behind attribute grammars (although they also highlighted some aspects to be improved, like the way of providing feedback). Finally, our preliminary educational effectiveness experiments provided positive evidence on the effectiveness of using interactive simulations compared to traditional *paper-and-pencil* study methods.

Currently we are working on solving some of the problems detected during system assessment, and, in particular, the enhancement of the feedback provided. For this purpose we plan to include a rule-based expert tutor to suggest clues to the students when they make a mistake. Also, we are planning to perform additional experiences with students, to compare this system with its predecessor, and to use the data collected to improve both systems.

5. ACKNOWLEDGMENTS

Projects and grants TIN2010-21288-C02-01, TIN2009-13692-C03-03 and EDU/3445/2011 partially supported this research. Our thanks to Rafael Fernández-López and Ángel Valero-Picazo for their contribution to a previous version of *Evaluators*.

6. REFERENCES

[1] Aho, A.V. 2008. Teaching the compilers course. ACM SIGCSE Bulletin, 40(4). 6-8.

[2] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. 2007. Compilers: principles, techniques and tools (2nd Edition), Addison-Wesley.

[3] Allestein, B., Yost, A., Wagner, P., Morrison, J. 2008. A query simulation system to illustrate database query execution. Proc. of the 39th SIGSE technical symposium on Comp. Science Education (SIGCSE'08).

[4] Bednarik, R., Gerdt, P., Miraftabi, R., Tukiainen, M.: Development of the TUP Model - Evaluating Educational Software. Proc. of the 4th IEEE Int. Conference on Advanced Learning Technologies (ICALT'04).

[5] Castro-Schez, J.J., Redondo, M.A., Gallardo, J., Jurado, F. 2012. Designing and developing software for educative virtual laboratories with language processing techniques: lessons learned in practical experiments. Journal of Research and Practice of Information Technology, 2012 (Special collection on Software Engineering for e-Learning), *in press*.

[6] Desherm, H. L., MacFall, R. L., Uti, N. 2002. Animation of Java linked lists. Proc. of the 33rd SIGCSE technical symposium on Comp. Science Education (SIGCSE'02).

[7] Fowler, M. 2011. Domain-Specific Languages. Addison-Wesley.

[8] García-Osorio,S., Gómez-Palacios, C., García-Pedrajas, N. 2008. A Tool for Teaching LL and LR Parsing Algorithms. Proc. of the 13th Annual Conference on Innovation and Technology in Comp. Science Education (ITiCSE'08).

[9] Grinder, M.T. 2002. Animating automata: a cross-platform program for teaching finite automata. Proc. of the 33rd SIGCSE tech. symp. on Comp. Science Education (SIGCSE'02).

[10] Kleppe, A. 2008. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley.

[11] Knuth, D.E. 1968. Semantics of Context-free Languages. Mathematical System Theory 2(2), 127-145.

[12] Mernik, M and Žumer, V. 2003. An Educational Tool for Teaching Compiler Construction. IEEE Transactions on Education, 46, 1, 61–68.

[13] Paakki, J. 1995. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computer Surveys, 27, 2, 196-255.

[14] Plaisant, C. 2004. The challenge of information visualization evaluation. Proc. of the working conference on Advanced visual interfaces (AVI'04).

[15] Rodríguez-Cerezo, D., Gómez-Albarrán, M., Sierra, J.L. 2011. From Collection of Exercises to Educational Games: A Process Model and a Case Study. Proc. of the 11th IEEE Int. Conf. on Advanced Learning Technologies (ICALT'11).

[16] Sierra, J.L., Fernández-Pampillón, A.M., Fernández-Valmayor. A. An environment for supporting active learning in courses on language processing. Proc. of the 13th annual conf. on Innovation and technology in Comp. Science Education (ITiCSE'08).

[17] Urquiza-Fuentes, J., Gallego-Carrillo,M., Gortazar-Bellas, F., Velázquez-Iturbide, J.A. 2006. Visualizing the symbol table. Proc. of the 11th annual SIGCSE conf. on Innovation and technology in Comp. Science Education (ITiCSE'06).

[18] Ware, C. 2006. Information Visualization Perception for Design (2nd Edition). Elsevier.

[19] White, T.M., Way, T.P. 2006. jFAST: a java finite automata simulator. Proc. of the 37th SIGCSE technical symposium on Comp. Science Education (SIGCSE'06).

6.6 Attribute grammars made easier: EvDebugger a visual debugger for attribute grammars

Cita completa:

Rodriguez-Cerezo, D., Henriques, P. R., Sierra, J. L. Attribute Grammars Made Easier: EvDebugger a Visual Debugger for Attribute Grammars. En SIIE'14: Proceedings of International Symposium on Computers in Education, 23-28. 2014.

Resumen original de la contribución:

Compiler construction courses are usually considered by the students as a difficult subject of the Computer Science degree. The main problem found by the students is to fully understand the theoretical concepts taught during the course and its practical application to build a compiler. In this paper, we present a platform for the development and debugging of language processors based on attribute grammar-oriented specifications. The main aim of this tool is to help students to design their own language processors, supported by the visual debugger included. The animations provided by EvDebugger show, in an attractive way, how the attribute evaluation process is performed. In this way, students are able to solve design problems, improve the effectiveness and efficiency of their language processors and understand their operation through experimentation and debugging provided with the software tool. Besides, we performed an assessment study with students of a Compiler Construction course whose results are presented and discussed in this paper.

Referencia de citas bibliográficas:

[2][5][6][19][23][26][31][34][44][72][95][109][119][120][167][130][124][136] [139][141][145][157][158][177][178][181]

Attribute grammars made easier: *EvDebugger*

A visual debugger for attribute grammars

Daniel Rodríguez-Cerezo
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9
28040 Madrid (Spain)
+34913947506
drcerezo@fdi.ucm.es

Pedro Rangel Henriques
Departamento de Informática
Escola de Engenharia
Universidade do Minho Campus de
Gualtar - 4710-057 Braga (Portugal)
+351-253604470
pedrorangelhenriques@gmail.com

José-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9
28040 Madrid (Spain)
+34913947548
jlsierra@fdi.ucm.es

***Abstract*— Compiler construction courses are usually considered by the students as a difficult subject of the Computer Science degree. The main problem found by the students is to fully understand the theoretical concepts taught during the course and its practical application to build a compiler. In this paper, we present a platform for the development and debugging of language processors based on *attribute grammar*-oriented specifications. The main aim of this tool is to help students to design their own language processors, supported by the visual debugger included. The animations provided by *EvDebugger* show, in an attractive way, how the attribute evaluation process is performed. In this way, students are able to solve design problems, improve the effectiveness and efficiency of their language processors and understand their operation through experimentation and debugging provided with the software tool. Besides, we performed an assessment study with students of a Compiler Construction course whose results are presented and discussed in this paper.**

***Keywords*— Education in Compiler Construction; Attribute Grammars; Debugger; Compiler Generator**

I. INTRODUCTION

The mainstream Computer Science (CS) curricula recommendations highlight language processor construction as an important aspect to be taught to students in CS [1]. Therefore, many CS education syllabi include a subject related to Compiler Construction. In these Compiler Construction courses, there is a balance between theoretical concepts and their practical application for building language processors [2]. Students are usually asked to design and develop a language processor as a final project. However the students usually face many problems to complete this project mainly because they are not able to properly orchestrate the theoretical and practical parts of the course to build the compiler. From our experience, we have observed that students usually do not pay much attention to the design details of the compiler, focusing directly on its implementation. When evaluating such practices, we found deficiencies that could easily be solved with a better design.

In a course on Compiler Construction, different techniques and formalisms for the design and specification of language processors are taught. Traditionally, attribute grammars (AGs) are used to describe the semantic processing carried out by language processors [14]. Ideally, the students should to design their own compilers using AGs, and validate those through formal methods. However, the students usually do not check the validity of their AG-based specifications, which leads to different problems in the development of the final implementation. When students have to create an AG design, the first and main problem they find is to decide what attributes (inherited or synthesized) they have to define. But after this step, the next problem is the definition of the correct semantic equations because they have a big difficulty to understand the dependencies among attributes, in particular their evaluation order. Thus, this paper focuses on this problem.

In order to help students to validate their AG based designs, we have developed a software tool for language specification based on AGs, called *EvDebugger*. *EvDebugger* includes a visual debugger that helps students to understand the evaluation process derived from their specifications and to identify the potential design errors, in an attractive and intuitive way. The functionality of this tool is based on our previous experiences developing simulators for teaching formalism of attribute grammars as *Evaluators* [20], *Evaluators 2.0* [19], and *EvLib* [18].

The rest of the paper is organized as follows. Section II provides the related work. Section III introduces the software tool *EvDebugger* describing its features and its operation. Section IV presents the results obtained from an assessment experience carried out with students of a Compiler Construction course. Finally, section V provides the conclusions and some lines of future work.

II. RELATED WORK

The software tool proposed in this paper is oriented to assists the students in the first steps of the process of understanding AGs, as well as in the development of the final project of the Compiler Construction course. The instructors of this subject use different strategies to propose a compiler to develop as final project of the course:

- Projects focused on a conventional programming language. This is the approach most commonly chosen by instructors of Compiler Construction courses. The selected language is a reduced programming language that included, for instance, a couple of basic data types,

978-1-4799-4428-6/14/$31.00 ©2014 IEEE

basic operations on these data types, control structures (such as loops and conditionals) and an abstraction mechanism (functions or procedures). Some prototypical languages used to support this approach are COOL [3] or MINIML [4].

- Small language processing projects. In this strategy students undertake the implementation of small compilers (or even concrete parts of a compiler) in order to focus them on the concepts recently taught in class. The works in [11] and [22] provide an in-depth discussion of this strategy.

- Domain-specific languages. The authors in [16] propose a strategy based on Domain Specific Languages (DSLs) where students have to develop a language processor for a DSL for a specific domain. In this case, the instructors propose a specific and attractive domain to the students, and plan different processing that students must develop. Thus, students will develop incrementally a complete language processor step by step. Other examples of this approach are given by DSLs for graph representation [24], simple figure drawing [21] or robot action programming [25].

- Analysis and debugging of processors of real programming languages. This approach is oriented to illustrate how a real compiler works, and it is represented by works like the described in [25], where the authors explain that the traditional project approach is not enough to teach all the concepts needed for the students to get a realistic assessment in the course.

However, it is worthwhile to notice how most of these strategies are focused on the implementation part, downgrading to a secondary concern the design part, or even taking this for granted. Due to these flaws in the different strategies presented, *EvDebugger* arises to support students in the design phase of the compilers, which will empower an implementation strategy more faithful to the language selected by the instructor.

To assist students in the completion of their course project, there are different software tools:

- Compiler -Compilers such as JavaCC [8], CUP [10], ANTLR [15], BISON [9], etc. These platforms are capable of generating language processors directly from the specification, provided as a *translation scheme*: a context free grammar augmented with semantic actions.

- Platforms that provide visual and interactive aids for the user. These platforms can be used to have a better understanding of the processing performed by the specified compiler. For instance, ANTLRWorks [6] is an environment for the ANTLR language. It provides a complete editor enriched with tools to detect different problems in the specification introduced by the user (e.g., non-determinism). The environment is able to offer different visual representations of the components of the compiler or the syntax tree of a sentence under processing, in order to enrich the debugging utility of the tool. In a similar way, VCOCO [17] is an extension of the COCO parser generator that generates a processor that presents visual simulation of the different parts of the generated compiler.

The main problem of Compiler-Compiler tools, from an educational point of view, is that they are aimed at the development phases of the compiler. Therefore, students need a solid design to obtain the proper translation schemes for the language processor that they want to develop. Enrichment of these tools with visual capabilities, in their turn, could be considered as specialized debuggers for the evaluation process associated with the kind of patterns corresponding to each translation tool. However, they do not provide facilities to work directly with the processor's design language.

On the other hand, there are tools for generating language processors from their AG-based specifications. As representative examples we consider LISA and PAG:

- LISA [12] is a platform that can generate language processors directly from an attribute grammar specification. The system also provides visual representations for each processing stage of the compiler, in order to provide to the students a rich feedback and a powerful debugger.

- PAG (*Prototyping with Attribute Grammars*) [23] is a tool to generate language processors from an AG specification. The generated processors make it possible to browse the attributed syntax trees in order to figure out how the semantic evaluation process proceeded.

While there is no doubt that this kind of tools could be useful to the students to refine the design of their compilers, the visual representations provided are static ones, and therefore they are not focused on the semantic evaluation process itself.

Finally, considering that *EvDebugger* is a visual debugger, various tools of this kind can be found in the field of other declarative frameworks. For instance:

- ViMer [7] a visual debugger for Mercury, a logic programming language, is able to show the evaluation process using a special kind of trees that adequately represents the non-deterministic character of this kind of languages.

- On other hand, JI.FI [5] is a visual debugger for Java applications specially designed to test concurrent Java programs. Among its features it can be highlighted the ability to debug the behavior of the concurrent programs in real-time, and a query processing engine able to interpret temporal queries.

- Finally, PM/IDE [13], a complete IDE in the field of planning domain models, includes features to debug and test automatic plans inside the models. The platform provides different visual debug features that help developers to understand the executions of the plans and induced changes in the modeled environment.

The different visual debugging tools presented above make apparent how the main recurrent feature of this kind of tools is that they are able to improve the understanding of the implementation, or the evaluation process, of their respective

languages by using visualizations and animations that allow the developer to comprehend at a glance their own programs.

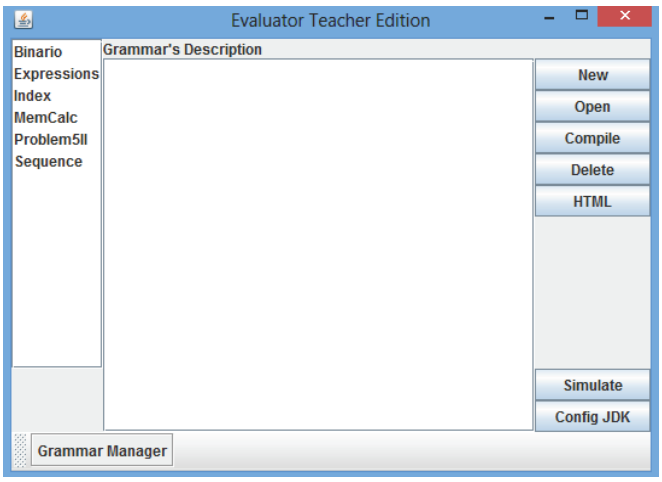


Fig 1 Snapshot of EvDebugger Grammar Manager.

III. EvDebugger

EvDebugger is a software tool for the specification of language processors. The tool is able to process AG specifications, described using a suitable specification language, and to generate a language processor for the language described. Also, the tool provides a visual debugger for the debugging of the language processor defined.

The next subsections describe the main components of *EvDebugger*: the *grammar manager*, the *grammar editor* and the *debugger*.

A. Grammar Manager

The grammar manager is the initial view offered by the tool. As can be seen in Fig 1, it offers a GUI divided into 3 areas. From left to right in Fig 1:

- The first area shows the different language processors registered in the system.
- The second area shows an informal description of the language (that must be provided by the language developer during creation) which is displayed when a language is selected in order to facilitate its identification.
- The third area provides access to the tool functionality, such as creating a new language, editing, deleting and compiling a previously created processor, and accessing the visual debugger using the *Simulate* button.

B. Grammar Editor

The grammar editor view is accessible from the grammar manager view after selecting the creation of a new language processor or opening a previously created one. Fig 2 shows a screenshot of this grammar editor. This view offers different areas to specify the different parts of a language processor: the attribute grammar specification (Fig 2a), the informal description of the processor (Fig 2b), and the semantic class (Fig 2c):

- The attribute grammar specification of the language processor must be written in a specification language that we created for this purpose. This specification language is very close to the notation usually used in Compiler Construction course lectures and books. In this way the specification of the language processor using an AG can be produced by students in a natural and easy way. The specification has three different sections: non-terminal symbols declaration, terminal symbols declaration, and the specification of the grammar rules.
- The informal description of the processor is a description of the language processor in a natural language to help the user to identify it in the grammar manager view.
- The semantic class is the name of a Java class that must implement the semantic functions used in the attribute grammar specification.

With these three components, the tool is able to automatically generate the language processor from the specification, which will be used by the debugger.

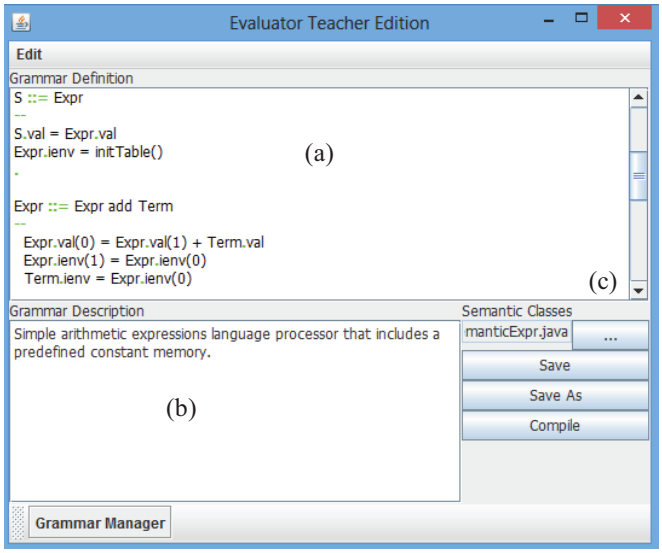


Fig 2 Snapshot of EvDebugger's Grammar Editor.

C. Debugger

To access the visual debugger, it is necessary to compile the (edited) grammar successfully in order to obtain a correct language processor. After selecting a compiled language processor, the developer can use the debugger clicking the *Simulate* button, available in the Grammar Manager GUI. Then, the debugger asks for a sentence of the language to process. If the sentence is accepted by the language processor, a debugger customized for the selected language and the sentence provided is displayed. Fig 3 shows the visual debugger fully customized.

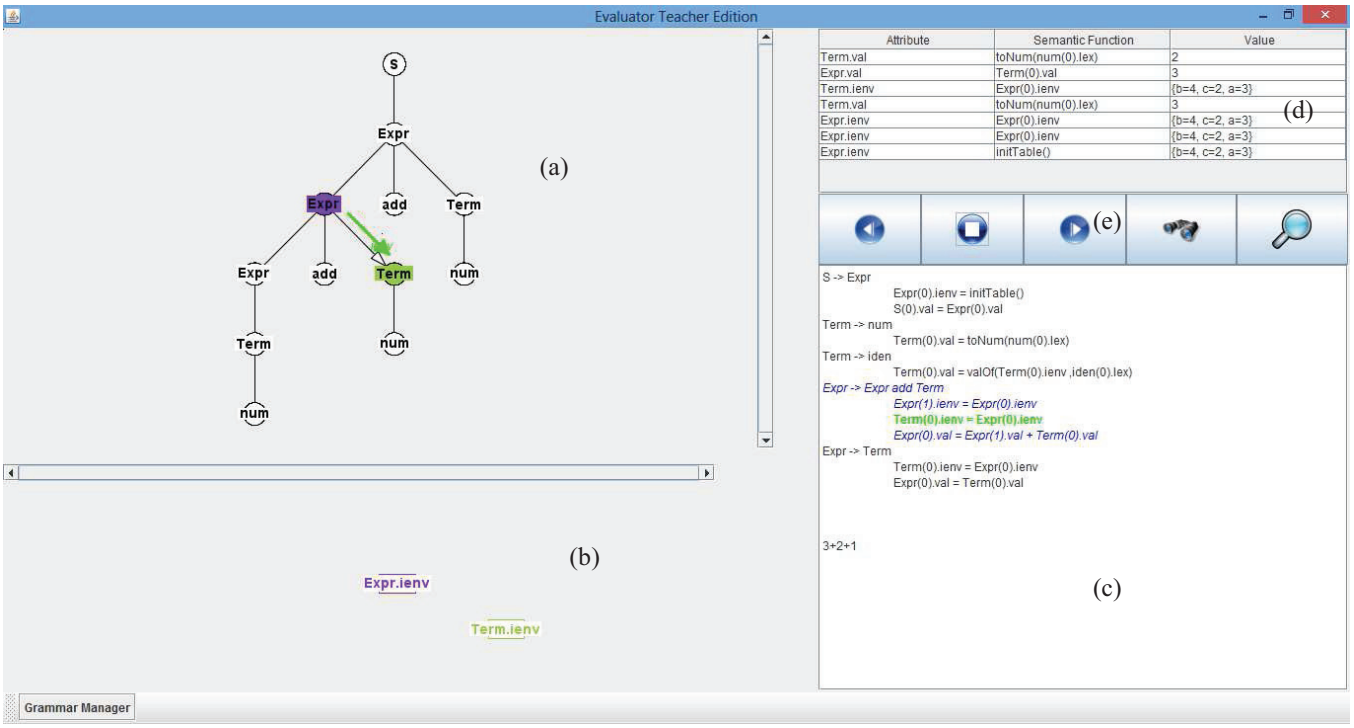


Fig 3 Screenshot of the visual debugger of the tool.

The debugger will display: (i) the syntax tree view (Fig 3a), (ii) the node attributes view (Fig 3b), (iii) the attribute grammar (Fig 3c), (iv) a table displaying the semantic values of the attributes computed (Fig 3d) and (v) the toolbox of the debugger (Fig 3e). In their turn, the different functionality of the toolbox's buttons are (from left to right): step-back debug button, play/stop debug button, step-forward debug button, node exploration button, and the syntax tree view modification button:

- The first three buttons are conceived to control the flow of the debug process.
- The node exploration button activates the node selection mode inside the syntax tree representation. When the user selects a tree node, the tool will draw the attribute instances associated with the node, below the syntax tree. These attributes can be selected, in order to check their value, when available.
- Finally, the syntax tree view modification button activates the visual exploration of the syntax tree. In this mode, the user can zoom in and out, in order to focus the view in the desired part of the syntax tree.

The visual debugger is able to display the evaluation process carried out by the AG specified and a sentence of the language. The debugger contemplates two different evaluation actions:

- Movements on the syntax tree. The movements through the tree nodes are shown to the user by the animation of a pointer that indicates which the node that is being currently visiting in the evaluation process is. When the evaluation process requires going to another node, the debugger animates the transition of the pointer to the destination node.
- Assignment of semantic values to instances of a semantic attribute.

In their turn, the assignment of semantic values in the debugger triggers the following animation sequence:

- In the syntax tree view, the nodes where the attribute instances involved in the semantic assignment are colored with different colors, and green arrows are used to indicate the flow of data associated with the assignment of this new semantic value in the syntax tree.
- In the attribute instance exchange view, the attribute instances involved in the evaluation action appears as boxes with different colors, corresponding with the color of the respective node, in the syntax tree. Also, in the attribute grammar specification, the syntax rule and the semantic equation involved are highlighted.
- Then, in the attribute instance exchange view, the different boxes converge on the box that represents the attribute instance whose value will be computed.
- Finally, when all the boxes have ended the animation, a new entry corresponding to the new semantic value calculated pops up in the table of the semantic values computed. The entry will register the attribute name, the semantic equation involved, and the semantic value computed.

The three main features described above compose *EvDebugger*, turning it into a platform for the development of

language processors from its specification in terms of the attribute grammar formalism. In addition, the visual debugger provided by *EvDebugger* is capable of displaying in an attractive way the semantic evaluation process for a sentence of the language processor selected. Thus, the developer is able to validate his/her design and identify potential errors and malfunctions in an easy and intuitive way.

IV. Assessment Study

During the 2013-2014 academic year, an experience with students of the subject *Compilers* at Universidade do Minho in the Master degree in Computer Engineering involving *EvDebugger* was carried out. In the experience participated 10 students who were asked to use the tool. The experiment was divided into the following phases.

- Presentation of the tool. *EvDebugger* functionality was showed through a live tutorial to the students.
- Use of the *EvDebugger*. At this point, we presented to the students different problems on language processing. The problems were designed to exploit all the semantic power of the attribute grammars. The language processing task proposed asked for students to design attribute grammars that required different degrees of inheritance.
- Collecting the opinions of students through a questionnaire. We designed a short survey containing nine questions with answers based on a four-point Likert-like scale (Disagree, Partially Disagree, Partially Agree, Agree). The questions are registered in the first column of Table 1, the second column shows the percentage of students that chose Agree and Partly Agree as an answer. We also provide a space for students to write freely their personal opinions about *EvDebugger*.

The results presented in Table 1 show that *EvDebugger* success has garnered the group of students surveyed. The assessment results obtained point that the functionalities provided by the editor and the debugger have been useful for students. The students consider both the grammar editor provided and the visual debugger for the creation and refinement of their language processor. Moreover, we have analyzed the different viewpoints expressed in the free questionnaire. Among the different opinions expressed, students pointed to various problems encountered during the experience. Most frequent suggestions have been the speed of the animations and the need for a mechanism for the rapid access to specific points during the evaluation process.

V. Conclusions and Future Work

Throughout this paper *EvDebugger* has been presented, a tool for the specification and debugging of language processors. The tool provides an editor for designing language processors based on the formalism of attribute grammars. Also, the platform contains a powerful visual debugger showing the evaluation process derived from the design. The educational power of this software lies mainly in its visual debugger and the animations provided that accurately illustrate the attribute evaluation process enhancing the dependencies among attributes. It allows students to better understand both the formalism and the operation of their own designs. Thus, students are able to refine the designs of their language processors and validate those before moving to the final deployment. The results of assessment included in this paper reveal that the perceived usefulness by students is high, since *EvDebugger* is considered as a complementary tool in the *Compilers* course, which is useful for understanding attribute grammars and their application to the design of language processors.

Table 1 Results obtained from the opinion survey.

Survey Items	% Agreement
Grammar Editor	
I found the attribute grammars editor easy to use	90
The visual aids of the editor (highlighting of reserved words and symbols) seemed useful to me	90
I think the implementation of semantic functions in a Java class is appropriate	90
The error messages provided by the compiler proved appropriate	30
Visual Debugger	
The visual debugger has helped me to improve the design of grammars posed	100
The animations presented by the debugger has helped me understand the data flow of my design	90
During animation, highlighting the rule syntactic and calculation involved in every step of debugging was helpful to understand my design	90
The arrows decorating the syntax tree during animations have helped me to understand the difference between inherited and synthesized attribute	80
The table where the calculated semantic attributes are stacked has helped me visualize the progress of the calculation process	90

Finally, as lines of future work it is firstly mentioned the need to fill the gaps found in *EvDebugger* by students during the assessment study: improving the error messages, and providing quick access to specific points in the evaluation process. Then, we would like to organize another experience with students to measure the educational effectiveness of the tool, and its impact on the design and development of the final project of the course.

Acknowledgment

This research was partially supported by grants TIN2010-21288-C02-01, EDU/3445/2011 and National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

REFERENCES

[1] ACM/IEEE. Computer Science Curriculum 2008: An Interim Revision of CS 2001. 2008.

[2] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques and tools (second edition), Addison-Wesley, 2007.

[3] Aiken, A. (1996). Cool: A portable project for teaching compiler construction. ACM SIGPLAN Notices, 31(7): 19-24, 1996.

[4] Baldwin, D. (2003). A compiler for teaching about compilers. *In SIGCSE'03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 220-223, New York, NY, USA. ACM.

[5] Blanton, E., Lessa, D., Arora, P., Ziarek, L., & Jayaraman, B. (2013). JI. FI: Visual test and debug queries for hard real-time. Concurrency and Computation: Practice and Experience.

[6] Bovet, J. and Parr, T. 2008. ANTLRWorks: an ANTLR grammar development environment. Softw: Pract. Exper., 38: 1305–1332. doi: 10.1002/spe.872.

[7] Cameron, M., Garcia De La Banda, M., Marriott, K., & Moulder, P. (2003, August). Vimer: a visual debugger for mercury. *In Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming* (pp. 56-66). ACM.

[8] Copeland, T. (2007). Generating parsers with JavaCC. Alexandria: Centennial Books.

[9] Donnelly, C., & Stallman, R. (2004). Bison. The YACC-compatible Parser Generator.

[10] Hudson, S. E., Flannery, F., Ananian, C. S., Wang, D., & Appel, A. W. (1999). Cup parser generator for java. Princeton University.

[11] Ledgard, H.F. 1971. Ten mini-languages: A study of topical issues in programming languages. *ACM Computing Surveys*, 3(3).115-146.

[12] Mernik, M. & Zumer, V. 2003.An educational tool for teaching compiler construction. IEEE Transactions on Education, vol.46. 61-68.

[13] Ong, J. C., Remolina, E., Smith, D. E., & Boddy, M. S. (2013). A Visual Integrated Development Environment for Automated Planning Domain Models.

[14] Paakki, J. 1995. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computer Surveys*, 27, 2, 196-255.

[15] Parr, T. (2009). ANTLR parser generator. Online Stand Dezember.

[16] Pereira, M. J., Oliveira, N., Cruz, D., & Henriques, P. (2013). Choosing grammars to support language processing courses. *Proceedings of the 2nd Symposium on Languages, Applications and Technologies*. OpenAccess Series in Informatics (OASIcs).

[17] Resler, D. & Deaver, D. 1998. VCOCO: A visualization tool for teaching compilers. *Proceedings of the 6th annual Conference on the Teaching of Computing and the 3rd annual Conference on Integrating Technology into Computer Science Education* (ITiCSE'98), ACM, New York, NY, USA. 199-202.

[18] Rodríguez-Cerezo, D. & Sierra, J-L. 2013. Introducing a Design-Preserving Implementation Strategy in a Compiler Construction Course. XV Simposio Internacional de Informática Educativa (SIIE'13). ACM. Pags 24-29.

[19] Rodríguez-Cerezo, D., Gómez-Albarrán, M. & Sierra, J.L. (2013). Interactive educational simulations for promoting the comprehension of basic compiler construction concepts. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (ITiCSE '13). ACM, New York, NY, USA, 28-33.

[20] Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M. & Sierra, J.L. (2014). Serious games in tertiary education: A case study concerning the comprehension of basic concepts in computer language implementation courses, *Computers in Human Behavior*, Volume 31, February 2014, Pages 558-570, ISSN 0747-5632.

[21] Ruckert, M. 2007. Teaching compiler construction and language design: making the case for unusual compiler projects with postscript as the target language. *Proceedings of the 29th SIGCSE Technical symposium on Computer science education*, ACM, New York, NY, USA.232-236.

[22] Shapiro, H.D. & Mickunas, M.D. 1976. A new approach to teaching a first course in compiler construction. *Proceedings of the ACM SIGCSE-SIGCUE technical symposium on Computer science and education*, ACM, New York, NY, USA.158-166.

[23] Sierra, J.L., Fernández-Pampillón, A.M. & Fernández- Valmayor. A. 2008.An environment for supporting active learning in courses on language processing. *Proceedings of the 13th annual conference on Innovation and technology in computer science education* (ITiCSE '08) 128-132.

[24] Werner, M. 2003.A parser project in a programming languages course. *Journal of Computing in Small Colleges*, 18(5).184-192.

[25] White, E., Sen, R. & Stewart, N. 2005. Hide and show: using real compiler code for teaching. *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, ACM, New York, NY, USA.12-16.

[26] Xu, L. & Fred, G. 2006. Chirp on crickets: teaching compilers using an embedded robot controller. *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, ACM, New York, NY, USA.82-86.

6.7 A systematic approach to the implementation of attribute grammars with conventional compiler construction tools

Cita completa:

Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools. Computer Science and Information Systems, 9(3): 983-1017. 2012.

Resumen original de la contribución:

This article describes structure-preserving coding patterns to code arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up and top-down parser generation tools. In these translation schemes, semantic actions are written in terms of a small repertory of primitive attribution operations. By providing alternative implementations for these attribution operations, it is possible to plug in different semantic evaluation strategies in a seamlessly way (e.g., a demand-driven strategy, or a data-driven one). The pattern makes possible the direct implementation of attribute grammar-based specifications with widely-used translation scheme driven tools for the development of both bottom-up (e.g. YACC, BISON, CUP) and top-down (e.g., JavaCC, ANTLR) language translators. As a consequence, initial translation schemes can be successively refined to yield final efficient implementations. Since these implementations still preserve the ability to be extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.

Referencia de citas bibliográficas:

[1][5][7][8][14][15][16][22][25][29][33][47][52][53][63][68][70][74][78][80]
[84][82][85][86][89][91][97][99][101][106][108][114][118][120][122][121]
[125][124][128][129][144][148][149][152][154][155][159][172][173][175][181]

DOI:10.2298/CSIS111223022R

A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools

Daniel Rodríguez-Cerezo[1], Antonio Sarasa-Cabezuelo[1], and José-Luis Sierra[1]

[1] Computer Science School,
Complutense University of Madrid
Calle Profesor José García Santesmases, s/n
28040 Madrid, Spain
{drcerezo, asarasa, jlsierra}@fdi.ucm.es

Abstract. This article describes structure-preserving coding patterns to code arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up and top-down parser generation tools. In these translation schemes, semantic actions are written in terms of a small repertory of primitive attribution operations. By providing alternative implementations for these attribution operations, it is possible to plug in different semantic evaluation strategies in a seamlessly way (e.g., a demand-driven strategy, or a data-driven one). The pattern makes possible the direct implementation of attribute grammar-based specifications with widely-used translation scheme-driven tools for the development of both bottom-up (e.g. YACC, BISON, CUP) and top-down (e.g., JavaCC, ANTLR) language translators. As a consequence, initial translation schemes can be successively refined to yield final efficient implementations. Since these implementations still preserve the ability to be extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.

Keywords: Attribute Grammars, Parser Generators, Language Processor Development Method, Grammarware

1. Introduction

Attribute grammars were introduced by Donald E. Knuth [25] as an extension of context-free grammars for describing the syntax and semantics of context-free languages, and are widely used as a high-level specification method for the first stages of the design and implementation of a computer language [2][35].

In order to make an attribute grammar-based specification executable, it is possible to use one of the many specialized tools that support the formalism

(see, for instance,[12][17][31][33][35]). However, regardless the recognized advantages of these tools, in practice, traditional implementations of language processors are rarely based on artifacts directly generated from attribute grammars. On the contrary, attribute grammars are taken as initial specifications of the tasks to carry out, while final implementations are usually achieved by using scanner and parser generators (e.g., ANTLR, CUP, Flex, Bison…), general-purpose programming languages, or a suitable combination of the two techniques [2]. The process of transforming the initial specification into a final implementation is usually ill-defined, and typically depends solely on the programmer's art –a programmer who many times discards formal specifications while he or she directly hacks the final implementation. It seriously hinders the systematic development and maintenance of language processors.

In order to bridge the gap between attribute grammar-based specifications and final implementations, we propose articulating the language processor development process as the explicit transformation of the initial attribute grammar-based specification to the final implementation. According to our proposal, the first step to convey during the implementation stage is to explicitly code the attribute grammar in the input language of the development tool (usually, a parser generator like Bison, CUP, JavaCC or ANTLR). This will make it possible to yield an initial running implementation, which subsequently could be refined to achieve greater efficiency. In addition, since the refined implementation still supports the explicit incorporation and subsequent refinement of attribute grammar-based features, the incremental development and subsequent maintenance of the language processor can be greatly facilitated. Therefore, it is important to notice that the rationale of the present work is not to provide new methods to automatically generate language processors from attribute grammars (in this case, undoubtedly the best choice would be one of the pre-existing tools based on attribute grammars). Instead, the rationale is to start from an attribute grammar specification and then to systematically refine it across several stages, finishing with a final, highly efficient implementation in a conventional compiler construction tool -a process which is not the aim of any typical attribute grammar tool.

This paper is mainly focused on the first step of our proposal, i.e. how to code an attribute grammar in terms of the input language supported by a conventional parser generation tool, although we also illustrate some aspects of the latter refinement. In order to cover the most widely used parser generation tools, we address both bottom-up parser generators of the YACC and CUP type and top-down parser generators of the JavaCC or ANTLR style. Unlike works in L-attributed [28] or LR-attributed grammars [4] and similar approaches (e.g., [23]), our approach will support the implementation of arbitrary non-circular attribute grammars. In addition, the coding pattern will be independent of the final evaluation style chosen. Indeed, attribute grammars will be coded by using a small repertory of *attribution* operations. Finally, by providing alternative implementations for these operations, it will be possible to set up the semantic evaluation style that will finally be used.

The structure of the rest of the paper is as follows: section 2 introduces some preliminaries. Section 3 details the dependency description operations and outlines two alternative implementations, which makes apparent how to plug in different evaluation styles. Section 4 describes the coding pattern for bottom-up parser generation tools. Section 5 describes the pattern for top-down ones. Section 6 presents some work related to ours. Finally, section 7 concludes the paper and outlines some lines of future work. A preliminary version of this work, which only deals with a former pattern for bottom-up translation schemes, can be found in [41].

2. Preliminaries

In this section we introduce some basic concepts concerning the two main language-processing specification tools addressed in this paper: attribute grammars (subsection 2.1) and translation schemes (subsection 2.2).

2.1. Attribute grammars

The formalism of attribute grammars was initially proposed by Donald E. Knuth at the end of the 1960s to characterize the semantics of context-free languages [25]. Attribute grammars introduce a syntax-directed, dependency-driven language processing style. This processing style is syntax-directed because the processing of each sentence is driven by its syntactic structure, and it is dependency-driven because it is directed by the dependencies among the computations involved. Figure 1 shows an example of an attribute grammar that models the evaluation of simple arithmetic expressions, followed by declarations of constants. In the formalized process, declarations are used to build an *environment* (a set of variable-value pairs), which is subsequently used to determine the value of variables. For the sake of conciseness, only the addition operator is considered.

Attribute grammars extend *context-free grammars* with *semantic attributes* and *semantic equations*. Indeed, *context-free* grammars are standard mechanisms to define the syntax of computer languages. In a context-free grammar:

- Syntax is defined by means of *syntax rules* (or *productions*), which determine the structure of syntactic constructions in terms of sequences of simpler constructions. For instance, in Figure 1 `Sent::= Exp` **`where`** `Decs` is a syntax rule that describes the top-level structure of the kind of sentences considered in this example.
- Syntactic constructions are represented by means of *syntax symbols*: composite structures by *non-terminal* symbols and simple structures by *terminal* symbols. For instance, in Figure 1 `Sent`, `Exp` and `Decs` are non-terminal symbols that represent, respectively, sentences,

expressions and declarations. In turn, **`where`**, **`var`** or **`num`** are terminal symbols (these symbols represent, respectively, the *where* reserved word, variables and numbers in the language considered).

- For each non-terminal there are one or several rules defining its structure. Each rule is made up of a *left-hand side rule* (*LHS*; the non-terminal whose structure is defined) and of a *right-hand side rule* (*RHS*; the sequence of symbols which define such a structure). For instance, the previously referred to rule established that a sentence (`Sent`, the rule's LHS) maybe (the rule's RHS): an expression (`Exp`), followed by the **where** reserved word, and followed by a block of declarations (`Dec`).
- There is also a distinct non-terminal (the grammar's initial symbol or the *grammar*'s *axiom*), which represents the language's highest level structure. In Figure 1, the grammar's initial symbol is `Sent`.

```
Sent ::= Exp where Decs
    Exp.env↓ = Decs.env↑
    Sent.val↑ = Exp.val↑
Exp ::= Exp + Opnd
    Exp1.env↓ = Exp0.env↓
    Opnd.env↓ = Exp0.env↓
    Exp0.val↑ = Exp1.val↑ + Opnd.val↑
Exp ::= Opnd
    Opnd.env↓ = Exp.env↓
    Exp.val↑ = Opnd.val↑
Opnd ::= num
    Opnd.val↑ = toNum(num.lex↑)
```

```
Opnd ::= var
    Opnd.val↑ = valOf(var.lex↑,Opnd.env↓)
Opnd ::= (Exp)
    Exp.env↓ = Opnd.env↓
    Opnd.val↑ = Exp.val↑
Decs ::= Decs , Dec
  Decs0.env↑ = extendWith(Dec.env↑,Decs1.env↑)
Decs ::= Dec
  Decs.env↑ = Dec.env↑
Dec ::= var = num
  Dec.env↑ = {(var.lex↑,toNum(num.lex↑))}
```

Figure 1. An example of attribute grammar

In a context-free grammar, syntax rules enable the description of the structure of each language's sentence in terms of a tree, which is called the *parse tree* of the sentence. Inner nodes are non-terminals, while leaves are terminals. Each parent node, together with its ordered sequence of child nodes, corresponds to the application of a syntax rule. Finally, the root node corresponds to the grammar's axiom. Figure 2a shows an example of sentence in the language considered in Figure 1, and Figure 2b shows the parse tree for this sentence. Notice how this tree makes the structure of the sentence explicit. Thus, subsequent processes can be driven by this structure.

As indicated before, an attribute grammar adds a set of *semantic attributes* to the symbols of an underlying context-free grammar. These attributes will take values in the corresponding nodes of the parse trees. Attributes can be of two types:

- *Synthesized attributes*: their values are computed from synthesized attributes in the owner node's child nodes and from the inherited attributes of this owner node. Thus, the value of a synthesized attribute represents (part of) the meaning of the symbol(s) to which this attribute is

associated. In the grammar of Figure 1, synthetized attributes are terminated with ↑. Thus, `val↑` is an example of synthetized attribute in this grammar, which is used to contain the values of operands (`Opnd` non-terminal), expressions (`Exp` non terminal) and sentences (`Sent` non-terminal). In turn, the synthesized attribute `env↑` is used to build the aforementioned environment from declarations. Finally, notice that terminal symbols can also have synthesized attributes; these synthesized attributes are called *lexical attributes*, and they should be set during lexical analysis. For instance, in the grammar of Figure 1 we use a lexical attribute, `lex↑`, which contains the actual string (the *lexeme*) of each token (e.g., for **`num`** it will contain the actual number, for **`var`** the actual variable, …).

(a) `x+y+5` **`where`** `x=5,y=6`

(b)

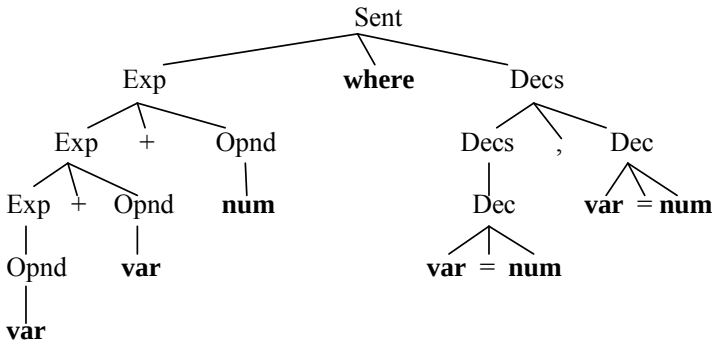


Figure 2. (a) A sentence of the language defined by the context-free grammar behind Figure 1, (b) Parse tree for the sentence in (a)

- *Inherited attributes*: their values are computed from inherited attributes in the parent and/or from synthesized attributes in the siblings. Thus, inherited attributes provide additional contextual information needed to determine the meanings of the symbols to which they are associated. In the grammar of Figure 1, we use an `env↓` inherited attribute to propagate the environment to the *expression* part of the input sentence, since this information is necessary to correctly determine the value of the constant appearing in such an expression part.

The attribute grammar will also add a set of *semantic equations* to each syntax rule. These equations will indicate how to compute the values of synthesized attributes in the rule's LHS, as well as the inherited attributes in the RHS symbols. More precisely:

- There will be exactly one semantic equation for each synthetized attribute on the LHS, and another one for each inherited attribute on the RHS.
- Each equation will apply *semantic functions* to other attributes in the rule. We will assume that, in the computation expressed by each equation, it

will only be possible to use inherited attributes from the LHS and synthesized attributes from the RHS (i.e., we will consider attribute grammars in Bochmann's normal *form* [9]).

For instance, the semantic equation $\texttt{Exp}_0\texttt{.val}\uparrow = \texttt{Exp}_1\texttt{.val}\uparrow + \texttt{Opnd.val}\uparrow$ for the syntax rule `Exp ::= Exp + Opnd` in the grammar of Figure 1 establishes that, in order to compute the value of a sum ($\texttt{Exp}_0\texttt{.val}\uparrow$)[1], it is necessary to add the value of the first operand ($\texttt{Exp}_1\texttt{.val}\uparrow$) to the value of the second operand ($\texttt{Opnd.val}\uparrow$).

Attribute grammars enable *semantic evaluation* on *attributed parse trees* (i.e., parse trees along with the semantic attributes for each node). Semantic evaluation is *dependency-driven*, since it is solely constrained by the dependencies that exist among these semantic attributes (i.e., to compute the value of an attribute, the only rule that must be obeyed is to have the values available of all the other attributes required by this computation according to a suitable semantic equation). Aside from this basic constraint, evaluation order does not matter. In consequence, attribute grammars result in a high-level specification formalism, since it is possible to specify language-processing tasks by focusing on the meaning of the syntax structures, without being distracted by lower-level implementation details, like the exact order in which attribute instances must finally be evaluated. In addition, the formalism is highly modular: it facilitates the addition of new attributes and semantic equations without affecting the existing ones, since the dependencies among attribute instances will be responsible for automatically rearranging the order in which to carry out the evaluation.

A convenient way of describing dependencies between attributes in an attributed parse tree is by means of a *dependency graph.* Nodes in this graph are the attributes in the symbols on the tree. Each arc denotes that the source attribute must be used to compute the value of the target one. Figure 3 shows the attributed parse tree and the dependency graph for the sentence in Figure 2a.

An attribute grammar is *non-circular* when it is not possible to find an attribute instance in a parse tree depending (directly or indirectly) on itself. For the contrary, the grammar is called a *circular* attribute grammar. Although semantic evaluation can be extended to manage circular attribute grammars (see, for instance [19]), for translation purposes non-circular attribute grammars usually suffice. Therefore, in this paper we will deal with non-circular attribute grammars. Semantic evaluation in these grammars can be meaningfully explained as follows [2]:

- First, find a topological order of the nodes in the dependency graph for the sentence being processed (since the grammar is non-circular, the

[1] Notice that, in order to refer to particular occurrences of a non-terminal symbol in a rule, it is possible to use subscripts: thus, Exp_0 refers to the first occurrence of Exp, Exp_1 to the second occurrence, etc.

dependency graph will be acyclical). In this order, each attribute instance will precede all the attribute instances depending upon it.

- Then, evaluate the attribute instances according to this order:

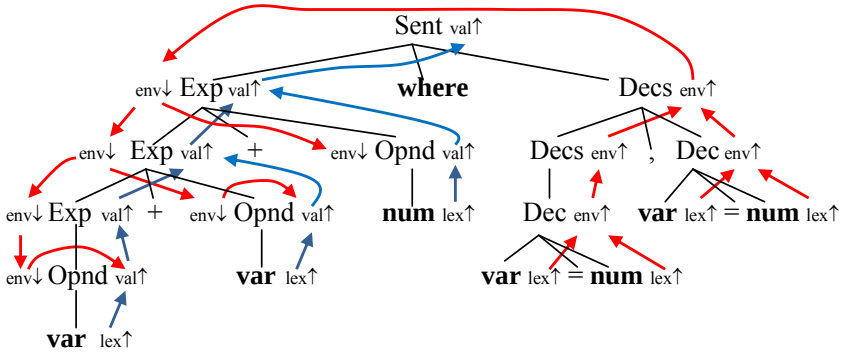


Figure 3. Attributed parse tree and dependency graph for the sentence in Figure 2a

However, it is only a conceptual execution model. In practice, semantic evaluation can be carried out by following different strategies which are only constrained by dependencies among attributes. Also, a particular evaluation strategy may not require the explicit construction of a parse tree. In fact, for remarkable subclasses of attribute grammars (many *s-attributed grammars*, which only involve synthesized attributes, and some classes of *l-attributed grammars*, in which inherited attributes of symbols only depend on the inherited attributes of their parents and synthesized attributes of their preceding siblings), it is possible to yield implementations that evaluate the attributes *on-the-fly* during parsing of the input sentence, without requiring the explicit construction of the syntax tree. Notice the grammar in Figure 1 is not s-attributed (it is needed to propagate the environment to the expression in order to evaluate it), nor l-attributed (because declarations are placed after the expression, and constant values are required to compute the value of such an expression).

2.2. Translation schemes

Translation schemes constitute another formalism that extends context-free grammar to allow the specification of syntax-directed processing [2]. For this purpose:

- Translation schemes adopt explicit visit orders for the nodes of the parse trees. Although many others are possible, two well-known visit orders are *left-to-right bottom-up* and *top-down* ones. In both of them child nodes are visited from left-to-right. However, in a bottom-up visit, nodes are visited in post-order, while in a top-down visit are visited in pre-order. In

addition, in a bottom-up visit order the visit to each node has only one *significant point*, once all its children have been visited. On the other hand, in a top-down one there are many significant points: (i) when the node is entered the first time, (ii) after a child has been exited and before the next one is entered, and (iii) when the node itself is exited.

- Translation schemes also adopt explicit ways of storing computed semantic information. For this purpose, it can be stored in semantic attributes, as in the case of attribute grammars, but also by using other means. For instance, typical execution models for bottom-up translation schemes use stacks for storing semantic information, while typical execution models for top-down ones assume implementations based on mutually recursive subprograms and use subprogram parameters and the runtime stack as a semantic storage mechanism. In addition, both bottom-up and top-down translation schemes can use global variables to facilitate some translation tasks.
- These artifacts conceive of the syntax rules as *visit plans*. For this purpose, they introduce a *semantic reference* mechanism to consult and update semantic information, as well as interleave chunks of code (*semantic actions*) at those points of the rule's RHS corresponding to significant visit points. Semantic actions will be executed each time the corresponding significant visit point is reached during the translation process. In particular, in bottom-up translators it will be possible to place a semantic action at the end of each syntax rule, while in top-down ones it will be possible to place semantic actions in any point of the rules' RHSs. In consequence, the latter will allow more natural translation patterns than the former. This is particularly true for the managing of inherited semantic information.

Although, in principle, translation schemes are independent of parser generation tools, as they can be conceived of as artifacts for processing parse trees, they are usually used as input specification formalisms for these tools. The resulting tree processors are then coupled with the parsing algorithms, and the explicit construction of the parse trees is definitively avoided. In particular:

- Bottom-up translation schemes are used as input to shift-reduce, LR parser generation tools of the YACC type (e.g., YACC, Bison, CUP, …). The resulting parsers use a stack to attach a semantic value to each syntax symbol, and they can also use global variables to manage additional semantic information. These tools constrain underlying context-free grammars to the LR type (usually, LALR(1) grammars) [2], although there are tools accepting more general grammars (e.g.,30).
- Top-down translation schemes are used as input to predictive descent parser generation tools of the JavaCC or ANTLR type. Since these tools

usually generate recursive descent parsers[2], semantic information is managed as parameters and return values of the subprograms generated, as well as in global variables, and the explicit construction of the parse tree is also avoided. These tools usually impose stronger constraints on the underlying context-free grammars: LL grammars. Although modern generation tools like ANTLR provide many useful extensions to basic LL(k) grammars (in particular, it supports the so-called LL(*) parsing method, which provides unbounded look-ahead enabled by finite-state predictors [37][38]), they are unable to manage features like left-recursion. However, as indicated before, they enable more natural mechanisms for dealing with inherited information.

Figure 4a shows an example of a bottom-up translation scheme. The language processed is the classical language of binary numbers proposed by Knuth in [25] to illustrate basic concepts in attribute grammars, and the processing task is to compute the values of the numbers. As in the other bottom-up translation schemes in this paper, we do not commit to any particular generation tool, and we do use a YACC-like notation [2] to refer to semantic values of symbols in the parse stack. Figure 4b shows a top-down, predictive-recursive translation scheme for this task. The underlying grammar is changed to LL(1), and the semantic actions are changed in consequence. Therefore, it will allow its implementation by using any of the mentioned top-down parser generation tools. As in the case of bottom-up translation schemes, we will not commit to particular generators. In addition, we will use ↓ to annotate input parameters and ↑ to annotate output ones.

(a)

```
Num ::= Num Bit {$$ := $1*2+$2}
Num ::= Bit {$$ := $1}
Bit ::= 0 {$$:=0}
Bit ::= 1 {$$:=1}
```

(b)

```
N(↑v) ::= Num(0,v)
Num(↓cv,↑v) ::= Bit(vb) RNum(vb,v)
RNum(↓cv,↑v) ::= Bit(vb) RNum(cv*2+vb,v)
RNum(↓cv,↑v) ::= {v := cv}
Bit(↑v) ::= 0 {v := 0}
Bit(↑v) ::= 1 {v := 1}
```

Figure 4. (a) An example of bottom-up translation scheme

3. The Attribute Evaluation Framework

Our coding pattern is largely based on the explicit description of the attribution structure of each grammar rule. For this purpose, we needed to develop an attribute evaluation framework, to be used in the semantic actions of the translation schemes. In this section we describe such a framework. For this purpose:

[2] It is also possible to generate non-recursive, table-driven descent parsers [2], but the mainstream in top-down parser generators is geared to the *recursive* model.

- Subsection 3.1 describes the set of basic *attribution operations* used in the translation schemes. These attribution operations make it possible to describe, for each syntax rule: (i) the dependencies between attribute occurrences in the symbols of this rule, and (ii) the functions to be used in order to compute the value of the attributes. They also make it possible to build *semantic contexts* for syntax rules (i.e., tables of references to attributes), to consult and set the value of individual attributes, and to control garbage collection.
- Subsection 3.2 introduces *semantic function managers* as the main extension points of the framework. Semantic function managers are the components used to execute semantic functions.
- Finally, subsections 3.3 and 3.4 describe two alternative implementations of the attribution operations, each based on a different *evaluation style* (a *demand-driven* style in subsection 3.3, and a *data-driven* one in subsection 3.4). In the demand-driven evaluation style, the values of attributes are computed in a lazy way, as they are required. On the other hand, in the data-driven style, values of attributes are computed in an eager way, as soon as the values of the attributes on which they depend become available. These implementations can be interchanged in a transparent way, without further changes in the translation schemes.

3.1. Attribution Operations

Table 1 outlines the repertory of basic attribution operations along with their intended meanings. As such a description makes apparent, the purpose of these operations is to provide the developer with the tools necessary to describe how the *attribute dependency graph* associated with a sentence can be built as this sentence is analyzed by the parser. In addition, it also lets the developer indicate the semantic functions for computing each attribute instance. It does not necessarily mean the graph must be fully stored in memory: depending on the actual implementation of the attribution operations, it will be possible to optimize, to a greater or lesser extent, the heap footprint, as the following subsections make apparent.

Table 1. Attribution operations

Operation	Intended Meaning
mkCtx(*n*)	It creates and initializes a *semantic context*: the list of attribute instances for a syntax symbol.
mkDep (a_0, a_1)	It sets a dependency between two attribute instances. Indeed, it declares that the attribute instance a_0 depends on the attribute instance a_1.
inst(*a*,*f*)	It *instruments* the attribute instance *a* by establishing *f* as the semantic function to be applied during evaluation (*f* is actually an integer identifier of such a semantic function)
release(*as*)	It invokes garbage collection on the attribute instance list *as*.
release(*a*)	It invokes garbage collection on the attribute instance *a*
set(*a*,*val*)	It fixes the value of the attribute instance *a* to *val*.
val(*a*)	It retrieves the value of the attribute instance *a*.

3.2. Semantic Function Managers

Before proceeding with the implementation of the attribution operations, it is convenient to introduce the concept of *semantic function manager*. In our approach, given a particular attribute grammar, the *semantic function manager* is an auxiliary component that supports the execution of semantic functions. Therefore, it is the main extension point of the evaluation framework, since it makes it possible to tailor it to each particular attribute grammar.

A semantic function manager can be conceived as a procedure that, taking the semantic function's identifier and the sequence of attribute instances as input, returns the result of applying the function to the attribute instances. It is important to remark that this component must be provided for each particular attribute grammar. Nevertheless, in our minimalistic conceptualization, we will assume this manager has the pre-established name `exec`. The implementation of this `exec` procedure will be changed from coding to coding[3].

As an example, Figure 5 depicts the pseudo-code for a semantic function manager for the grammar in Figure 1. Notice that, for each equation it is necessary to: (i) substitute attribute references in the equation's RHS for values of the semantic function manager's attribute arguments (e.g., Exp_1`.val`↑ `+ Opnd.val`↑ becomes `val(ARGS[0])+ val(ARGS[1])`, and (ii) associate a suitable integer number to the underlying semantic function (e,g., the `ADD` constant in Figure 4).

```
def IDEN=0; def ADD=1; def TONUM=2; def VALOF=3;
def EXTEND=4; def SINGLEENV=5;

procedure exec(FUN,ARGS) {
case FUN of
  IDEN →
   return val(ARGS[0]);
  ADD →
   return val(ARGS[0])+ val(ARGS[1]);
  TONUM →
   return toNum(val(ARGS[0]));
  VALOF →
   return valOf(val(ARGS[0]),val(ARGS[1]));
  SINGLEENV →
   return {( val(ARGS[0]), toNum(val(ARGS[1])) ) }
  EXTEND →
   return extendsWith(val(ARGS[0],val(ARGS[1]))
end case
}
```

Figure 5. Semantic function manager for the attribute grammar in Figure 1

[3] Although it is possible to achieve more elegant solutions by using a programming language with minimal higher-order support (e.g., a conventional object-oriented language), our conceptualization is deliberately maintained as simple as possible to preserve the essence of the evaluation approaches.

3.3. Demand-Driven Evaluation

According to the *demand-driven* evaluation style, semantic evaluation starts once the sentence has been completely parsed (see, for instance [18][29]). At this point, there is an in-memory representation of the part of the dependency graph required for performing semantic evaluation. During evaluation, the values of the attribute instances will be calculated only when they are required. For the sake of simplicity, we will ignore the detection of potential circularities in the underlying dependency graphs, although it would not be difficult to extend the framework to support it.

The first step in setting this implementation is to decide how to represent semantic attributes. For this purpose, the instances of the semantic attributes can be conceived as records. Table 2 outlines the fields required together with their intended purposes. Thus, this representation makes it possible to build a dependency structure in which:

Table 2. Structure of attribute instances in the demand-driven evaluation framework.

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	$\perp$
available	A boolean flag that indicates whether the value is available.	`false`
deps	It keeps the links to those attribute instances required to compute the value.	*The empty list*
semFun	It stores the integer code of the semantic function required to compute the value.	$\perp$
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

- Each attribute instance points to those attribute instances required to compute it (in a similar way to the *reversed* dependency graph used in [18]).
- In addition, it explicitly stores the identifier of the semantic function to be used in this computation.

Once this representation is decided, it is possible to proceed with the coding of the operations themselves. Table 3 outlines it using pseudo-code. In this pseudo-code, references are intended to work as in Java, although we do not assume automatic garbage collection (instead, a `delete` primitive is explicitly invoked). Indeed, this is why we explicitly include `release` attribution operations.

The different operations behave as follows:

- `mkCtx` collects, in a list, as many fresh attribute instances as needed. This list actually represents a *semantic context* for a syntax symbol, since it gives access to all its semantic attributes.
- `mkDep` adds the second attribute instance in the `deps` list of the first one.
- `inst` stores the semantic function code in the `semFun` field.
- `release`, when applied to a list of semantic attribute instances, releases each instance and de-allocates the list itself.

- On the other hand, when `release` is applied to an attribute instance, it decreases its reference count by 1. If this count reaches 0, the instances on which it depends are released; finally, the original instance itself is de-allocated.
- `set` sets the `value` field and records its availability.
- `val` recovers the value of an attribute instance as follows: (i) if the value is available, it returns such a value, (ii) otherwise, it calls the semantic function manager to compute such a value and sets and returns it.

Table 3. Implementation of the attribution operations to allow a demand-driven evaluation style

Operation	Implementation	Operation	Implementation

mkCtx(*n*)	*as* := **new list**
	for i := 1 **to** *n* **do**
	add(*as,* **new attribute**)
	end for
	return *as*
		release(*a*)	*a*.refcount:= *a*.refcount-1
			if *a*.refcount = 0 **then**
			foreach *a'* **in** *a*.deps **do**
			release(*a'*)
			end foreach
			delete *a*.deps
			delete *a*
			end if

mkDep (a_0, a_1)	**add** (a_0.deps, a_1) a_1.refcount := a_1.refcount + 1	set(*a*,*val*)	*a*.value := *val* *a*.available := **true**

inst(*a*,*f*)	*a*.semFun := *f*	val(*a*)	**if** ¬ *a*.available **then**
			set(*a*,
			exec(*a*.semFun,*a*.deps))
			release(*a*.deps)
			end if
			return *a*.value

release(*as*)	**foreach** *a* **in** *as* **do** release(*a*) **end foreach** **delete** *as*		

Thus, the demand-driven evaluation process arises from the interplay of the `val` attribution operation and the semantic function manager. Also notice how explicit garbage collection can be readily interleaved in the implementation of the attribution operation by appropriately managing the reference counters and by de-allocating lists and records as soon as they become unreachable. Although in this evaluation style, most of the dependency graph remains in memory until parsing is finished, automatic garbage collection makes it possible to de-allocate useless parts of the graph when they become unreachable. This can be due to attribute instances that are not ultimately required in any computation, or to successive evolutions of the implementation, combining pure attribute grammar features with implementation-oriented optimizations (e.g., global variables, on-the-fly evaluation of semantic attributes, …).

3.4. Data-Driven Evaluation

In the *data-driven* evaluation style, attribute instances are scheduled to be evaluated as soon as the values for all the instances on which it depends are available (see, for instance, [24]). Thus, this method can shorten the duration of attribute instances. Additionally, it can interleave evaluation with parsing. These features can be of interest while processing very long sentences, or sentences made available asynchronously (e.g., on a network communication channel). However, this method can do useless evaluations on attribute instances not required to yield the final results.

Table 4 outlines the representation of attribute instances in this case. Notice that, in addition to the list of instances on which an instance depends, the reverse relationship needs to be maintained (i.e., each attribute instance must refer to those instances which depend on it). Indeed, this representation is similar to that used by networks of *observables-observers* in the *observer* object-oriented pattern [14][4].

Table 4. Structure of attribute instances in the data-driven evaluation framework

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	⊥
available	A boolean flag that indicates whether the value is available.	`false`
deps	It keeps the links to those attribute instances required to compute the value.	*The empty list*
obs	It keeps the links to those attribute instances *observing* it (i.e., which depend on it to compute their values).	*The empty list*
required	Counter which records the number of attribute instances in *deps* whose values have not yet been determined.	0
semFun	It stores the integer code of the semantic function required to compute the value.	⊥
instrumented	*True* if *semFun* was set, *false* otherwise.	`false`
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

Table 5 outlines the pseudo-code of the attribution operations whose implementation differs from those in the demand-driven style. This way, we only need to redefine `mkDep`, `inst`, `set` and `val`:

- In addition to updating `deps` in the first instance, `mkDep` must test whether the second instance has already been computed. If it is not available, the first instance must be added to its `obs` list, since such an instance depends on its value, which is not yet available.
- Note `inst` must take care of whether the value can be computed. Indeed, if the corresponding attribute instance has all the instances on which it depends computed, it can thereby be computed. It assumes the

[4] As with the demand-driven style, this representation could be simplified by inferring the values of flags (in this case, *available* and *instrumented*) from the other fields. However, we prefer to explicitly preserve these flags to increase the readability of pseudo-code.

establishment of all the required dependencies before instrumentation, which is ensured by our coding pattern.
- `Set` must take care to decrement the `required` counters in all the instances depending on the current one. In addition, if a counter reaches 0, it must force the evaluation of the corresponding instance.
- Finally, `val` immediately computes the value, unless the instance has not yet been instrumented.

Notice how, in this case, evaluation can be interleaved with parsing. Indeed, evaluation is fired when the values of attribute instances are explicitly set, and also when attributes are instrumented. In consequence, garbage collection also interplays with parsing, and, therefore, this method can mean less heap usage. However, this method assumes all the semantic functions used are *strict,* in the sense that all their arguments must be evaluated before they are applied. On the contrary, the demand-driven method described in the previous subsection also supports *non-strict* functions, in which the way of evaluating the arguments can differ from function to function.

Table 5. Implementation of the attribution operations to allow a data-driven evaluation style (only those implementations differing from Table 3 are presented)

Operation	Implementation	Operation	

mkDep (a_0, a_1)	**add** (a_0.deps, a_1)
	a_1.refcount := a_1.refcount + 1
	if ¬ a_1.available **then**
	  add (a_1.obs, a_0)
	  a_0.required := a_0.required + 1
	  a_0.refcount := a_0.refcount + 1
	end if
		set(a,*val*)	a.value := *val*
			a.available := **true**
			foreach a' **in** a.obs **do**
			  a'.required := a'.required – 1
			  if a'.required = 0 **then**
			  val(a')
			  end if
			end foreach
			release(a.obs)

inst(a,f)	a.semFun := f
	a.instrumented := **true**
	if a.required = 0 **then**
	  val(a)
	end if
		val(a)	**if** ¬ a.available ∧
			  a.instrumented **then**
			  set(a, exec(a.semFun,a.deps))
			  a.available := **true**
			  release(a.deps)
			end if
			return a.value

4. A Coding Pattern for Bottom-up Parser Generation Tools

In this section we introduce a coding pattern for bottom-up parser generation tools. In this way:
- In order to keep the translation scheme as independent as possible of changes in the attribute grammar's semantic part, we will promote an intermediary representation of the attribute grammar based on *attribution functions* (subsection 4.1). For this purpose, with each rule will be assigned a function that takes the semantic contexts of the rule's RHS as arguments and builds and returns the semantic context for the rule's LHS. In addition,

using the basic attribution operations introduced in the previous section, attribution functions establish dependencies among attributes, associate semantic functions with attributes as necessary, and control garbage collection.
- Then, these functions will be used in the actions of the resulting bottom-up translation scheme (subsection 4.2). More precisely, the semantic action associated with each rule will invoke the attribution function for this rule with the suitable set of arguments.
- The analysis of the memory footprint required by the overall method will be depicted in subsection 4.3 by considering both the demand-driven and the data-driven evaluation styles.
- Finally, subsection 4.4 briefly illustrates some potential refinements of the initial implementation. These refinements will be oriented to anticipate the computation of inherited attributes by using *marker non-terminals* (i.e., new non-terminals defined by rules with empty RHS), and to simplify implementation by means of global variables.

4.1. The attribution functions

The implementation of the attribute grammar using a bottom-up parser generation tool can be naturally thought of as the *bottom-up* construction of the attribute dependency graph for each processed sentence using basic attribution operations. In this construction, the dependency graph for a syntactic structure is built by taking the dependency graphs of the substructures as building components. Thus, the process can be facilitated by introducing a set of *attribution functions*, which, for each rule in the grammar, take cares of this construction. These attribution functions will be used to set up the semantic actions of the bottom-up translation scheme that feeds the parser generation tool. Therefore, the set of attribution functions can be conceived of as the implementation of a sort of *abstract* version of the attribute grammar, which subsequently can be attached to a concrete syntax by using a suitable translation scheme.

Each attribution function takes the semantic contexts of the symbols in the rule's RHS as input, and it outputs the semantic context for the LHS non-terminal using basic attribution operations. In order to do so, it is possible to apply the following guidelines:
- First at all, we need to create the semantic context for the LHS. This is done by using an `mkCtx` operation. We only need to indicate the number of semantic attributes for the LHS non-terminal.
- Next, we need to describe the dependencies among the attribute instances. Such dependencies are directly determined by examining the semantic equations, and they must be stated by using the `mkDep` operation.
- Once this has been done, it is necessary to *instrument* the synthesized attribute instances in the rule's LHS, as well as the inherited attribute

instances of the RHS symbols. Once more, the code is straightforward: an `inst` operation for each equation. Notice we need to code the semantic functions with integer identifiers, which can be interpreted by the semantic function manager.

- Finally, we need to release the attribute instance lists for the symbols in the rule's RHS.

This process can be further facilitated by using a procedure establishing the corresponding dependencies for each attribute as well as the instrumentation. This procedure, which will be called `eq` (since it actually serves to represent semantic equations), is sketched in Figure 6. Finally, notice that, although we need to provide an attribution function for each rule in the grammar, the same function can be shared by several rules. Therefore, in addition to contributing to more readable translation schemes, attribution functions also make it possible to reuse common attribution patterns. Indeed, it is possible to provide attribution functions with additional parameters in order to increase the reuse degree.

```
procedure eq(lhsAtr,rhsAtrs,semFun) {
   foreach rhsAtr in rhsAtrs
     mkDep(lhsAtr,rhsAtr)
   end foreach
   inst(lhsAtr,semFun)
}
```

Figure 6. The `eq` procedure

As an example, Figure 7 depicts the attribution functions for the attribute grammar in Figure 1. For instance, the `addition` function codes the attribution for the rule `Exp ::= Exp + Opnd` in the grammar of Figure 1 as follows:

- Since `Exp`, the rule's LHS, has two semantic attributes (`env` and `val`), we need to invoke `mkCtx` with 2 as the number of attributes to be allocated.
- From the first equation, we get Exp_1`.env` depends on Exp_0`.env`. In addition, the semantic function to be applied is the identity. Therefore, the equation is coded by `eq(Exp1[env], (Exp0[env]),IDEN).`
- The other equations are coded in a similar manner. For instance, the equation Exp_0`.val = `Exp_1`.val + Opnd.val` is coded by `eq(Exp0[val], (Exp1[val],Opnd[val]),ADD).` Notice that, for each equation, it is important to establish the dependencies in the order in which the attribute references appear in its RHS, and therefore it must be taken into account in the coding of each equation.
- Finally, we include a `release` action for each symbol in the rule's RHS having semantic attributes.

Concerning the allocation of lexical attribute instances, it must be performed by the scanner, which will return the corresponding attribute instance list using a suitable field in the token.

```
def env=0; def val=1; def vs=0; def lex=0;
function init(Exp,Decs) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   eq(Exp[env],(Decs[env]),IDEN)
   release(Exp)
   release(Decs)
   return Sent
 }
function addition(Exp1,Opnd){
   Exp0 := mkCtx(2)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp0[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp1)
   release(Opnd)
   return Exp0
 }
 function chain(Child) {
  Parent := mkCtx(2)
  eq(Child[env],(Parent[env]),IDEN)
  eq(Parent[val],(Child[val]),IDEN)
  release(Child)
  return Parent
}
function num(num) {
   Opnd := mkCtx(2)
   eq(Opnd[val], (num[lex]),TONUM)
   release(num)
   return Opnd
 }
function var(var) {
   Opnd := mkCtx(2)
   eq(Opnd[val],
       (var[lex],Opnd[env]),VALOF)
   release(var)
   return Opnd
 }
function mutiEnv(Dec,Decs1) {
  Decs0 = mkCtx(1)
  eq(Decs0[env],
     (Dec[env],Decs1[env]),EXTEND)
  release(Dec)
  release(Decs1)
  return Decs0
}
function singleEnv(Dec) {
  Decs = mkCtx(1)
  eq(Decs[env],(Dec[env]),IDEN)
  release(Dec)
  return Decs
}
function entry(var,num) {
  Dec = mkCtx(1)
  eq(Dec[env],
      (var[lex],num[lex]),SINGLEENV)
  release(var)
  release(num)
  return Dec
}
```

Figure 7. Attribution functions for the attribute grammar in Figure 1

4.2. The bottom-up translation scheme

In order to finish the coding, it is necessary to provide a suitable translation scheme. It can be done in a straightforward way, by using the attribution function that corresponds to each rule. Indeed, for each syntax rule A ::= α in the grammar, we only need to add a rule A::= α {\$\$:= ϕ($\$_\alpha$)} to the translation scheme. Here, ϕ is the attribution function for A::= α, and $\$_\alpha$ denotes the list of RHS semantic contexts. This pattern makes further advantages to using attribution functions apparent, instead of directly coding the semantic equations in the rule's actions (like we did in our previous work [41]): the concrete syntax can be readily changed without changing the attribution functions (which, as indicated before, are actually the implementation of an abstract version of the original attribute grammar).

Figure 8 exemplifies the coding pattern by showing the bottom-up translation scheme that implements the attribute grammar of Figure 1. Coded in the input language of a tool like YACC, Bison or CUP, and with a suitable implementation of the attribution functions and the basic attribution operations, it can be automatically turned onto a running implementation.

```
Sent ::= Exp where Decs      {$$ := init($1,$3)}
Exp ::= Exp + Opnd           {$$ := addition($1,$3)}
Exp ::= Opnd                 {$$ := chain($1)}
Opnd ::= num                 {$$ := num($1) }
Opnd ::= var                 {$$ := var($1) }
Opnd ::= (Exp)               {$$ := chain($2) }
Decs ::= Dec, Decs           {$$ := multiEnv($1,$3)}
Decs ::= Dec                 {$$ := singleEnv($1)}
Dec  ::= var = num           {$$ := entry($1,$3)}
```

Figure 8. Bottom-up translation scheme for the attribute grammar in Figure 1

4.3. Analysis of the method

The efficiency of the language processor generated will be manifested in the memory footprint of the recognition and evaluation process, which will in turn depend on the evaluation strategy used and on the kind of the initial attribute grammar:

- If the implementation uses the demand-driven evaluation style, it will incur in the highest amount of auxiliary memory required by the method. Indeed, the memory usage will be rather independent of the kind of the grammar, and proportional to the length of the input sentences. Indeed, the dependency graph will be almost entirely built before evaluation is initiated, and the process will be divided into two well differentiated phases: (i) a first one in which the input sentence is recognized and the dependency graph is built, and (ii) a second one in which the attribute values are computed.
- If the implementation uses the data-driven evaluation style, the performance will be optimal for s-attributed grammars. Indeed, the values of the attributes will be computed as soon as they are instrumented, and the amount of additional memory required for semantic evaluation will remain constant. However, in the presence of inherited information, the evaluation will be delayed until this information is *injected* into the process. The worst case happens when the overall evaluation process depends on inherited information to be set up in the grammar's initial symbol. In this case, most of the dependency graph must be built before initiating evaluation, and thus the method becomes equivalent to using a demand-driven strategy.

This analysis does not mean the method does not provide good (even nearly optimal) solutions for non s-attributed attribute grammars, since inheritance is not required to be global. For instance, for grammars like that of the example, the method, in combination with a data-driven evaluation style, yields not only nearly optimal, but also elegant implementations.

4.4. Refinements

Once the initial coding is available, the initial implementation can be systematically refined in an efficient implementation by using well-known techniques for dealing with inherited information during bottom-up parsing. In particular:

(a)

```
Sent ::= Mo Exp where Decs  {$$ := init($1,$2,$4) }
Mo ::=                      {$$ := mkEnv() }
Exp ::= Exp + Opnd          {$$ := addition($1,$3)}
Exp ::= Opnd                {$$ := chain($1)}
Opnd ::= num                {$$ := num($1) }
Opnd ::= var                {$$ := var($1,$0) }
Opnd ::= (M1 Exp)           {$$ := chain($3) }
M1 ::=                      {$$ = $-1}
Decs ::= Dec, Decs          {$$ := multiEnv($1,$3)}
Decs ::= Dec                {$$ := singleEnv($1)}
Dec ::= var = num           {$$ := entry($1,$3)}
```

(b)

```
...
function mkEnv() {
   return mkCtx(1)
}
...
function init(ExpEnv,Exp,Decs) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   eq(Exp ExpEnv[env],(Decs[env]),IDEN)
   release(ExpEnv)
   release(Exp)
   release(Decs)
   return Sent
}
...
```

```
function addition(Exp1,Opnd){
   Exp0 := mkCtx(2 1)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp1[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp1)
   release(Opnd)
   return Exp0
}
...
function var(var, Env) {
   Opnd := mkCtx(2 1)
   eq(Opnd[val],
       (var[lex],Opnd Env[env]),VALOF)
   release(var)
   return Opnd
}
```

Figure 9. (a) Refinement of the translation scheme in Figure 8 by means of marker non-terminals; (b) modification of some attribution functions and the addition of a new one (erased code appears in strikethrough light-gray text, and new added coded appears shaded)

- Use of *marker non-terminals* (i.e., new non-terminal symbols defined by empty rules [2]) to mark the beginning of *left spines* (i.e., chains of elements generated by left-recursion). These non-terminals can store inherited attributes to which can be accessed from any point of the left spines without requiring explicit propagation. Using this technique, it is possible to deal with many l-attributed grammars with bounded memory footprint. The technique can be applied to the implementation exemplified before, yielding the translation scheme of Figure 9a. In this refinement it is possible to eliminate the inherited environment, since it can be remotely stored in the marker symbol **`Mo`** and referred from the marker symbol **`M1`**. In addition, the marker contexts can be passed on as an additional argument to the `var` attribution function. In Figure 9b we show the new attribution function `mkEnv` and how the old attribution functions `init`,

`addition` and `var` must be modified to fit in the new refinement. The other attribution functions can be modified in an analogous way, and therefore they will be omitted here.

- Use of global state. In order to integrate this global state in the evaluation machinery, it is possible to create *views* of this state as semantic attributes. The technique can be illustrated with the example discussed above, since the environment can be completely managed as a global variable. Thus, all the machinery concerning propagation of environments can be completely eliminated. Figure 10a shows the resulting translation scheme. Notice how the environment is managed as a global variable, and is also exposed as a globally accessible semantic attribute. With the exception of `init` (see Figure 10b), the attribution functions coincide with those used in the refinement sketched in Figure 9

(a)
```
global env = ∅
global aenv = mkCtx(1)
procedure addEntry(env,Var,Num) {
  env := extendWith({(val(var[lex]),
                      toNum(val(Num[lex])))},env)
}

Sent ::= Exp where Decs  {set(aenv[env],env); $$ := init($1); release(aenv); }
Exp ::= Exp + Opnd       {$$ := addition($1,$3)}
Exp ::= Opnd             {$$ := chain($1)}
Opnd ::= num             {$$ := num($1) }
Opnd ::= var             {$$ := var($1,aenv) }
Opnd ::= (Exp)           {$$ := chain($2) }
Decs ::= Dec, Decs       {}
Decs ::= Dec             {}
Dec  ::= var = num       {addEntry(env,$1,$3)}
```

(b)
```
function init(Exp) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   release(Exp)
   return Sent
}
```

Figure 10. (a) Use of a global environment to simplify the translation scheme in Figure 8; (b) the `init` attribution function in this refinement.

5. A Coding Pattern for Top-Down Parser Generation Tools

This section describes the coding pattern for top-down parser generation tools. For this purpose, it follows a similar structure to that of the previous one:

- Subsection 5.1 describes the structure of attribution functions in this pattern. In one sense, these attribution functions arose by *reversing* the bottom-up ones. Now, each attribution function takes the semantic context

of the LHS as argument, and it builds and returns the semantic contexts for each symbol in the RHS. As in the bottom-up cases, they also use the basic attribution operations to set up all the attribute evaluation machinery.
- Subsection 5.2 describes the general guidelines to code the translation scheme. As in the bottom-up case, it is carried out by placing attribution functions at strategic points in the syntax rules.
- Subsection 5.3 describes how to deal with underlying non-LL grammars. Indeed, bottom-up parser generation tools usually deal with predictive grammars of the LL-type, in which it is possible to determine which rule to expand by using a finite amount of input look-ahead. However, some grammatical features (e.g., left-recursion, common left-factors) destroy this capability to predict the rule to be applied. Fortunately, many of these grammars can be systematically transformed to forms suitable for top-down parsing. These transformations must be accompanied by the transformation of the semantic part, however. Thus, we researched how to perform these transformations for the case of our encoding scheme.
- As in the bottom-up case, subsection 5.4 briefly analyzes the method, and subsection 5.5 describes some subsequent refinements (the most prominent one deals with the systematic replacement of recursion by iteration in the resulting translation schemes).

5.1. The attribution functions

Although it is possible to undertake implementation by thinking of the bottom-up construction of the attribute dependency graph, as in the bottom-up case, it is possible to obtain more advantages if we think of the *top-down* construction of this graph. In particular, it will facilitate the propagation of inherited information during parsing.

```
function addition(Exp0){
   Exp1 := mkCtx(2)
   Opnd := mkCtx(2)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp0[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp0)
   return (Exp1,Opnd)
 }
```

Figure 11. Top-down geared version of the attribution function `addition`

To enable the top-down construction of the dependency graph, we need to reverse the flow of semantic contexts in the attribution functions. Now, these functions will take the LHS context as input and it will return the RHS contexts as output. Thus, a typical attribution function begins by creating the RHSs contexts. Then it establishes the dependencies between attributes and instruments the attributes as in the bottom-up case. Finally, it releases the LHS context. Figure 11 exemplifies it by showing the top-down geared

version of the `addition` attribution function. The other attribution functions can be adapted in a similar way, and therefore they will be not detailed here.

5.2. The top-down translation scheme

As in the bottom-up case, the coding of the translation scheme is carried out in terms of the attribution functions. In addition, due to the inversion of the flow of semantic contexts in the attribution functions, it is necessary to connect the terminal contexts created in these functions to the contexts created by the scanner. This can be done by using the `conn` procedure sketched in Figure 12 (the name is an abbreviation for *connect*).

```
procedure conn(termCtx,lexCtx) {
   eq(termCtx[lex],(lexCtx[lex]),IDEN)
   release(termCtx); release(lexCtx)
}
```

Figure 12. Procedure for connecting terminal contexts.

Thus, for each syntax rule $A::=X_0 .. X_n$ in the grammar, we need to add a rule $A(\downarrow ctxA)::=\{(ctx_0,\ldots,ctx_n):=\phi(ctxA)\ \}\ l_0\ \ldots\ l_n$ where: (i) ϕ is the rule's attribution function, (ii) $(ctx_0,\ldots,ctx_n)$ collects the RHS contexts (this assignment is optional; it can be omitted if the attribution function does not return any context), and (iii) each l_i is $X_i(ctx_i)$ if X_i is a non-terminal, $X_i(lexctx_i)$ {`conn(`ctx_i`,`$lexctx_i$`)`} if it is a terminal with semantic charge, or X_i if it is a terminal without semantic charge (a keyword, a punctuation symbol, etc.). These guidelines are illustrated in Figure 13, which shows the top-down translation scheme for the grammar in Figure 1.

```
Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓co)  ::= {(c1,c2) := addition(co)} Exp(c1) + Opnd(c2)
Exp(↓co)  ::= {c1 := chain(co)} Opnd(c1)
Opnd(↓co) ::= {c1 := num(co) } num(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := var(co) } var(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := chain(co) } (Exp(c1))
Decs(↓co)  ::= {(c1,c2) := multiEnv(co)} Dec(c1) , Decs(c2)
Decs(↓co) ::= {c1 := singleEnv(co)} Dec(c1)
Dec(↓co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 13. Top-down translation scheme for the attribute grammar in Figure 1 (warning: this translation scheme is not yet implementable with a top-down parser generator!)

Unfortunately, since top-down translators usually require LL underlying context-free grammars, translation schemes obtained according to the stated guidelines can require further transformation before allowing their implementation during parsing. In particular, the context-free grammar of the translation scheme in Figure 1 exhibits left-recursion, which make this coding

unsuitable for top-down parser generation. Next subsection deals with this problem.

5.3. Factoring and immediate left-recursion elimination

In many cases the problematic top-down translation schemes and the associated attribution functions can be systematically tuned by applying similar patterns to the well-known factoring and left-recursion elimination transformations presented in any compiler construction textbook [2]. In particular:

- Figure 14a sketches a transformation pattern for removing common factors in a rule-set. Notice this transformation supposes the explicit construction of the common factor's semantic context. It will be carried out by a *context-construction* function (denoted by $\texttt{mkCtx}_\alpha$ in Figure 14a). In addition, it is necessary to keep this context alive, regardless whether it will be released in the common factor. For this purpose, we need to create another twin context ($c_{\alpha p}$ in Figure 14a), and to connect it to the actual common factor's semantic context. This connection is achieved with a *context connection procedure*, denoted by $\texttt{conn}_\alpha$ in Figure 14a. Finally, it will require explicitly modifying the attribution functions for each rule affected. The modified attribution functions (denoted by ϕ'_i in Figure 14a) do not need to create the semantic context for the common factor; instead, they will take it as a parameter.
- Figure 14b shows a transformation pattern for removing immediate left-recursion. The pattern requires the explicit construction of the context for the recursive non-terminal, which is achieved by using a context-construction function ($\texttt{mkCtx}_A$ in Figure 14b). As usual, the chain generated by left-recursion in the original grammar is generated by using right-recursion in the transformed one. Each stage of this right-recursive process can be associated with a stage in the bottom-up construction of the parse tree in the original grammar. Therefore, it is possible to take the context associated to the root of the already constructed sub-tree as input, and then to modify the corresponding attribution function to take this as an additional argument instead of creating it (the modified functions are noted ϕ'_i in Figure 14b, and they must take care of releasing the semantic context once they are not necessary). In addition, it is necessary to provide a context connection procedure for performing the connection between the input and the last context created once the right-recursion is finished (it is denoted by $\texttt{conn}_A$ in Figure 14b).

Figure 15 illustrates the application of these patterns to the translation scheme of Figure 13. The grammar of the transformed scheme is LL(1) and, therefore, suitable for its implementation in any of the top-down parser generation tools mentioned.

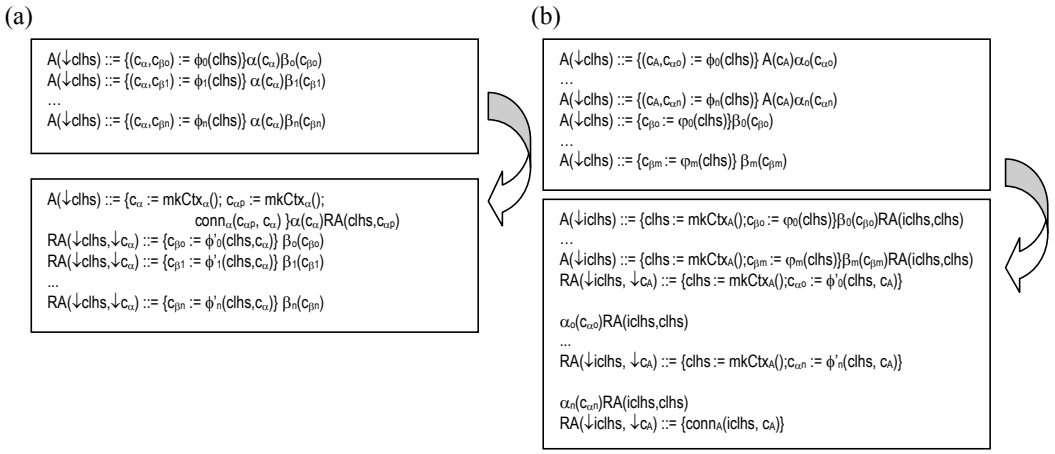


Figure 14. (a) Factoring pattern; (b) Immediate left-recursion elimination pattern

```
function mkCtxExp() {return mkCtx(2)}
function mkCtxDec() {return mkCtx(1)}
procedure connExp(ic,c) {eq(c[env],(ic[env),IDEN); eq(ic[val],(c[val]),IDEN) }
procedure connDecs(cp,c) {eq(cp[env],(c[env),IDEN);}

Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓ic)  ::= {co := mkCtxExp(); c1 := opnd(co)} Opnd(c1) RExp(ic,co)
RExp(↓ic, ↓c1) ::= {co := mkCtxExp(); c2 := addition(co,c1)} + Opnd(c2) RExp(ic,co)
RExp(↓ic, ↓co) ::= {connExp(ic,co)}
Opnd(↓co) ::= {c1 := num(co) } num(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := var(co) } var(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := chain(co) } (Exp(c1))
Decs(↓co)  ::= {c1 := mkCtxDec(); c1p := mkCtxDec(); connDecs(c1p,c1) }
                                                    Dec(c1) RDecs(co,c1p)
RDecs(↓co, ↓c1) ::= {c2 := multiEnv(co,c1)} , Decs(c2)
RDecs(↓co, ↓c1) ::= {singleEnv(co,c1)}
Dec(↑co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 15. Result of eliminating common factors and immediate left-recursion in the top-down translation scheme of Figure 13 (the transformed parts are shadowed) in order to obtain an artifact implementable with a top-down parser generator.

5.4. Analysis of the method

As in the bottom-up case, the use of a demand-driven evaluation style will imply explicitly constructing dependency graphs, and therefore the highest memory overhead. As in bottom-up implementations, it can be alleviated by using data-driven evaluation. In this case, the method will incur in the lowest auxiliary evaluation memory overhead for l-attributed grammars. Indeed, for these grammars, data-driven evaluation will yield a behavior equivalent to a one-pass, on-the-fly translation process.

Finally, since the initial coding encourages the explicit coding of the plain, BNF grammar, the resulting translators will be highly recursive, which should be taken into account if the final implementation language does not support tail recursion optimization. Fortunately, as will be indicated in the next section, by using EBNF notation in the underlying context-free grammars, it will be possible to easily turn many right-recursions into iteration.

5.5. Refinements

As in the bottom-up case, it is possible to use global state to simplify the propagation of context. Nevertheless, due to the nature of top-down translators, this refinement is less critical from a performance perspective. Concerning the use of marker non-terminals, it is nonsense in this scenario.

However, as indicated in the previous subsection, an interesting refinement would be to exploit the support of EBNF notation provided by typical predictive recursive parser generation tools in order to overcome the potential stack overflow problem associated with the recursive implementation of genuinely iterative processes[5]. Indeed, it is equivalent to performing a tail-recursion optimization process by hand[6].

In addition, it is possible to carry out several simplifications oriented to minimizing the use of temporary variables (e.g., by passing complex expressions as parameters to non-terminal symbols).

```
Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓ico)  ::= {co := mkCtxExp()} Opnd(chain(co)) RExp(ico,co)
RExp(↓ic, ↓c1) ::= ({co := mkCtxExp()} + Opnd(addition(co,c1)) {c1:=co})*
                                                        {connExp(ic,c1)}
Opnd(↓co) ::= num(lc1) {conn(num(co),lc1)}
Opnd(↓co) ::= var(lc1) {conn(var(co),lc1)}
Opnd(↓co) ::= ( Exp(chain(co)) )
Decs(↓co)  ::= {c1 := mkCtxDec(); c1p := mkCtxDec(); connDecs(c1p,c1) }
                                                    Dec(c1) RDecs(co,c1p)
RDecs(↓co, ↓c1) ::= ({co := multiEnv(co,c1)} ,
                     {c2 := mkCtxDec(); c1 := mkCtxDec(); connDecs(c1,c2) }
                      Dec(c2))* {singleEnv(co,c1)}
Dec(↑co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 16. Refinement of the translation scheme in Figure 15

Figure 16 exemplifies the result of applying these refinements on the translation scheme of Figure 15. The resulting scheme can be readily implemented on any typical recursive predictive parser generation tool (e.g., JavaCC or ANTLR), or directly by hand in a general-purpose programming language. As this example makes apparent, after applying this refinement,

[5] Notice this problem does not affect bottom-up parsers, provided sequences are represented by means of left-recursion.

[6] Indeed, it could be possible to directly formulate the immediate left-recursion elimination pattern in iterative terms.

recursion will only be used to express nesting (in the example, it is due to the use of parenthesis in expressions), which constitutes the most natural use of this grammar feature.

6. Related Work

As indicated in the introduction, the standard way of implementing an attribute grammar is to use one of the tools that directly supports the formalism. Indeed, as [35] makes apparent, since its invention by Knuth at the end of the sixties of the past century, the computer language community has proposed many of these tools, starting with classical systems like GAG [22], FNC-2 [20], ELI [15] or Elegant [7], and ending with recent proposals like LISA [17][31][33], Silver [51] or JastAdd [29]. These tools take attribute grammars as input, and generate operative language processors as output. In addition, they support metalanguages by adding many extensions to the basic formalism (e.g., modules [21], generics [42], higher-order [48], object [16] and aspect orientation [39][40], etc.), which facilitate the production and maintenance of complex specifications.

Attribute grammar-based systems as the abovementioned promote orchestrating the development entirely in terms of attribute grammars, and, in particular, in terms of the metalanguages supported. On the contrary, the goal of our approach is not to provide yet another attribute grammar system, but to propose systematic ways of integrating attribute grammars in conventional language implementation processes, by using conventional parser generation tools. In this way, in our approach attribute grammars are used at the initial stages of the development process, as a formal specification tool. In addition, our work promotes an initial design-preserving coding in a conventional parser generation tool, in the form of a suitable translation scheme. Beyond this point, the development process proceeds through several refinements, making use of the parser generation tool facilities and the tool's target implementation language.

In consequence, our approach promotes straightforward coding patterns, which can be applied by hand to get initial codings, and which make it possible to identify the different pieces of the original attribute grammar in these codings. On the other hand, the code generated by an attribute grammar-based tool is usually a highly optimized artifact, usually generated following a *static* approach in which evaluation and storage strategies are determined as the result of a static analysis of the input grammar [1], and which is not intended to be inspected and modified by humans.

In addition, our approach is oriented to converge with conventional development processes. Because of it, on one hand we encourage the use of semantic evaluation methods that can be easily coupled with parsing. This is not necessarily true for attribute grammar-based tools, many of which promote final implementations that operate on (concrete or abstract) syntax trees. Of course the patterns described in this paper could be automated in

the form of attribute-grammar based tools. Indeed, tools for the processing of XML based on attribute grammars like those described in [43] are inspired by these patterns (in particular, these tools use the data-driven evaluation strategy to make the stream-oriented, asynchronous, processing of very wide XML documents possible). These tools could be used as a sort of CASE support during the development process model promoted in this paper, which in turn could imply the provision of some roundtrip support (see the future work description in the next section).

The coupling of attribute evaluation and parsing has been extensively addressed as a way of implementing restricted classes of attribute grammars (see, for instance, [3] for a tutorial introduction). The works in [2][3] show how l-attributed grammars with underlying LL grammars can be implemented during top-down predictive descent parsing. In addition, different classes of LR-attributed grammars have been identified, which allow semantic evaluation to be implemented using straightforward extensions of LR parsers [4]. In the marriage of attribute grammars and logic programming, the class of *logical one-pass logical* attribute grammars shows how some kinds of right dependencies can also be managed during conventional top-down parsing [34][36]. Contrary to the work presented in this paper, all these approaches constrain the classes of allowed grammars to strict subclasses of non-circular attribute grammars. In contrast, our approach is able to deal with arbitrary non-circular attribute grammars. If the grammars are of certain types (e.g., l-attributed grammars with an LL(1) underlying context-free grammar), and a suitable semantic evaluation approach is used (e.g., a data-driven strategy), our implementations produce artifacts comparable in performance and memory footprint to those promoted by the abovementioned works. In other cases, the approach is still able to produce running implementations, which can adapt the memory footprint to that required for performing semantic evaluation.

The development of some attribute grammar-based systems has exploited the marriage between attribute grammars and parser generation tools. A common strategy is to build a preprocessor by translating an attribute grammar-based specification language into a running implementation written in terms of a parser generator. In [23] one of these systems is described, which takes an attribute grammar-like specification as input, and it turns it into a YACC implementation. However, since the resulting implementation evaluates attributes during parsing, the class of supported grammars is restricted to a subset of the LR-attributed ones. The Ox system [8] follows a similar approach, but it supports arbitrary non-circular attribute grammars. For this purpose, the processors generated decouple parsing and semantic evaluation by using an optimized implementation of the processing models behind attribute grammars (i.e., to build the parse tree, to arrange attribute instances in topological order, and then to perform evaluation according to this order). XLOP [43], a system developed by us to describe XML processing tasks as attribute grammars, also translates attribute grammar specifications into inputs to a parser generation tool (in this case, CUP). RIE [44], a system that supports a very general class of LR-attributed grammars (ECLR-

attributed grammars [4]) adopts a different implementation approach, by basing the metagenerator on an explicit modification of the Bison parser generation tool. Regardless of the implementation strategy followed (in these examples, based on preprocessors for / extensions to parser generation tools), they ultimately fall in the category of attribute grammar-based tools. Therefore, the general considerations made above concerning the relationships between our approach and attribute grammar – based tools also applies here.

Concerning parser generators, there is a plethora of systems available that can be used during the development of a language processor. A basic feature differentiating them is whether they generate top-down parsers (e.g., the aforementioned tools JavaCC [26] and ANTLR [38], as well as classic tools like COCO/R [32]), or bottom-up ones (e.g., the aformentioned YACC [45],Bison [27] and CUP [5], as well as tools like Tatoo [11], SableCC [13], Beaver[7], Copper [49] or YaJco[8]). Also, these tools differ in the class of grammars allowed (e.g., JavaCC supports LL(k) grammars, while ANTLR supports the aforementioned LL(*) parsing method, able to deal with unbounded look-ahead; additionally tools like Elkhound [30], SDF [10] or, under certain settings, Bison, provide support to arbitrary context-free grammars via the GLR parsing method [46]), by the expressiveness of its specification language (e.g., ANTLR or Tatoo support very sophisticated features, like grammar modularization, rule inheritance, etc.), by whether they include support for lexical specification (e.g., JavaCC, ANTLR) or whether it must be made by using a separating tool (e.g., CUP), and by many other features whose detailed analysis is beyond the scope of the present work. As was indicated, the patterns presented in this paper are applicable to most of these parser generators (in particular in those tools that support deterministic grammars; in tools like SDF, whose outcome is parse forests that must be subsequently disambiguated, the applicability of these patterns vanishes). Also, it is important to notice that, while many of these parser generation tools support the concept of *semantic attribute*, like attribute grammars (e.g., this terminology is explicitly included in ANTLR), it does not mean that these tools give direct support for attribute grammars. Indeed, in addition to managing semantic attributes, the essential aspect of attribute grammars is the support for a dependency-driven execution style: semantic evaluation is not necessarily coupled with parsing, but emerges as a consequence of the dependencies among attributes. In this way, the patterns introduced in this work make it possible to incorporate this computation style into specifications for parser generation tools, and, in consequence, to facilitate the subsequent refinement into more efficient implementations.

Finally, as the implementations of our attribution operations make apparent, we avoid the explicit construction of the parse tree. While this construction is necessary in order to support more sophisticated evaluation

[7] http://beaver.sourceforge.net/

[8] http://code.google.com/p/yajco/

strategies (see, for instance [1]), our simple coding patterns make it unnecessary, since it is centered directly on the construction of dependency graph-like structures. A similar technique is followed in [6], an implementation of circular attribute grammars in Prolog whose semantic equations are described by using λ-expressions. The execution model of the resulting artifact works in two stages: (i) construction of λ-expressions for the root's synthetized attributes, and (ii) interpretation of these expressions according to a least fixpoint semantics to yield the final values. Thus, the resulting approach resembles our demand-driven implementation. In [50], Prolog is also used to implement attribute grammars, and two evaluation strategies are proposed. The first one supposes building terms representing semantic expressions for the root's synthetized attributes, which are subsequently interpreted with a separate interpreter. The second one promotes the use of Prolog co-routine facilities to delay evaluation of arguments until they are instantiated. Thus, the first strategy is analogous to our demand-driven implementation (nevertheless, our implementation is optimized to avoid duplicated evaluations; see [47] for a similar implementation in Prolog that also avoids redundant evaluations). The second one is a Prolog implementation of a data-driven strategy.

7. Conclusions and future work

This paper has shown how to systematically code arbitrary non-circular attribute grammars in the input languages of bottom-up, LALR(1) parser generation tools like YACC, BISON or CUP, as well as top-down, LL parser generation tools like JavaCC or ANTLR. It is done by using a small set of attribution operations. These operations, in turn, can be implemented in different ways in order to enable different semantic evaluation styles. In particular, this paper has illustrated two alternative implementations: one supporting a demand-driven style, and another supporting a data-driven one. The results of this work can be useful to promote a systematic method of using conventional parser generation tools to yield final implementations. This method starts with the initial coding of an attribute grammar-based specification, and then it evolves it in a final implementation by applying systematic implementation patterns and techniques. Thus, by applying and documenting systematic refinements, it is possible, on one hand, to yield efficient implementations and, on the other hand, to track the refinement chain from these final implementations to the original attribute grammar-based specifications. Besides, the method facilitates the incremental introduction of new language features, since they can be described according to attribute grammar conventions, then readily coded in the implementation, and finally optimized according to implementation-dependent criteria. Therefore, the method transports the attribute grammar amenability to doing modular and extensible specifications incrementally to an implementation process based on parser generation tools.

Currently we have successfully tested our method with several small examples, and we are applying it to the development of a non-trivial translator for a Pascal-like language. From these experiences, we have realized how the encoding patterns are simple enough to being applied without specific tooling support (although, of course, this support could be a very valuable facility in our methodology). Also, we have gained further evidence on the feasibility and usefulness of our method with its application in an introductory compiler construction course during the first period of the 2011-2012 academic year at the Complutense University. Indeed, we proposed that our students produce initial implementations of language processors by taking attribute grammar specifications as a guide, and using the method described in this paper. We observed that they didn't find it more difficult to apply than students of previous courses found while hand-coding conventional recursive descent translators. In addition, the quality of the final programs was substantially better than in previous years, since the method encouraged rigorous adherence to the original specification. Thus, we plan to further apply it as a systematic learning method in future editions of the course. Also, as future work, we plan to provide the aforementioned tooling support in order to facilitate the application of the method: automatic application of the coding patterns to produce the initial translation schemes, support for some of the transformations and refinements described in this paper, roundtrip support and support for tracking successive refinements, and support for profiling and debugging the semantic evaluation processes.

Acknowledgements. Thanks are due to project grants TIN2010-21288-C02-01 and Santander-UCM GR 42/10, group reference 962022. Also, Daniel Rodriguez-Cerezo was supported by the Spanish University Teacher Training Program (EDU/3445/2011).

References

1. Ablas, H. Attribute Evaluation Methods. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science Vol. 545, Springer, 48-113. (1991)
2. Aho A.V, Lam M.S, Sethi R, Ullman J.D.: Compilers: principles, techniques and tools (2nd Edition). Addison-Wesley. (2006)
3. Akker, R., Melichar, B., Tarhio, J. Attribute Evaluation and Parsing. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 187-214. (1991)
4. Akker, R., Melichar, B., Tarhio, J.: The Hierarchy of LR-attributed grammars. In Deransart, P., Jourdan, M (eds.): Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA'90), Paris, France, Lecture Notes in Computer Science 461, Springer, 13-28. (1990)

5. Appel, A.W. Modern Compiler Implementation in Java. Cambridge University Press. (2002)
6. Arbab, B. Compiling Circular Attribute Grammars into Prolog. IBM Journal of Research and Development, Vol. 30, No. 3, 294-309. 1986
7. Augusteijn, L. The Elegant Compiler Generator System. In Deransart, P., Jourdan, M (eds.): Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA'90), Paris, France, Lecture Notes in Computer Science 461, Springer, 238-254. (1990)
8. Bischoff, K.M. Design, Implementation, Use and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex and C. TR #92-31, Dp. Of Computer Science, Iowa State University, (1992)
9. Bochmann, G.V.: Semantic Evaluation from Left to Right. Communications of the ACM, Vol. 19, No. 2, 55-62. (1976)
10. Brand, M.G.J v.d., Deursen, A, v., Heering, J., Jong, H.A.d., Jonge, M.d., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, JJ., Visser, E., Visser, J. The Asf +Sdf Meta-environment: A Component-Based Language Development Environment. In Wilhelm, R (ed.): Compiler Construction - Proceedings of the 10th International Conference on Compiler Construction CC'01, Genova, Italy, Lecture Notes in Computer Science, 2027, Springer, 365-370. (2001)
11. Cervelle, J., Forax, R., Roussel, G. Tatoo: an innovative parser generator. 4th International Symposium on Principles and Practice of Programming in Java PPPJ'06, Mannheim, Germany, ACM, 13-20. (2006)
12. Ekman, T., Hedin, G. The JastAdd system - modular extensible compiler construction. Science of Computer Programming, Vol. 69, No. 1-3, 14-26. (2007)
13. Gagnon, E.M., Hendren, L.J. SableCC, an Object-Oriented Compiler Framework. International Conference on Technology of Object-Oriented Languages TOOLS'98, Sta Barbara, CA, USA, IEEE, 140-154. (1998)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. (1995)
15. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM, Vol. 35, 121-131. (1992)
16. Hedin, G. An Object-Oriented Notation for Attribute Grammars. 3rd European Conference on Object-Oriented Programming, Nottingham, UK, Cambridge University Press, 329-345. (1989)
17. Henriques, P.R., Varanda-Pereira, M.J., Mernik, M., Lenic, M., Gray, J.G., Wu, H. Automatic Generation of Language-Based Tools using the LISA System. IEE Proceedings – Software, Vol. 152, No. 2, 54-69. (2005)
18. Jalili, F.: A general linear-time evaluator for attribute grammars. ACM SIGPLAN Notices, Vol. 18, No. 9, 35-44. (1983)
19. Jones, L.G.: Efficient Evaluation of Circular Attribute Grammars. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, 429-462. (1990)
20. Jourdan, M., Parigot, D.: Internals and Externals of the FNC-2 Attribute Grammar System. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 485-504. (1991)
21. Kastens, U., Waite, W.M.: Modularity and Reusability in Attribute Grammars. Acta Informatica, Vol. 31, No. 7, 601-627. (1994)

22. Kastens, U.: GAG: A Practical Compiler Generator. Lecture Notes in Computer Science 141, Springer. (1982)
23. Katwijk, J.: A preprocessor for YACC or a poor man's approach to parsing attributed grammar. ACM SIGPLAN Notices, Vol. 18, No. 10, 12-15. (1983)
24. Kennedy, K., Ramanathan, J.: A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing. ACM Transaction of Programming Languages and Systems, Vol. 1, No. 1, 142-160. (1979)
25. Knuth, D. E.: Semantics of Context-free Languages. Mathematical System Theory, Vol. 2, No. 2, 127–145. (1968). See also the correction published in Mathematical System Theory, Vol. 5, No. 1, 95–96.
26. Kodaganallur, V. Incorporating language processing into Java applications: a JavaCC tutorial. IEEE Software, Vol. 21, No. 4, 70-77. (2004)
27. Levine, J. Flex & Bison: Text Processing Tools. O'Reilly Media. (2009)
28. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed Translations. Journal of Computer and System Sciences, Vol. 9, No. 3, 279-307. (1974)
29. Magnusson, E. Hedin, G.: Circular Reference Attributed Grammars—Their Evaluation and Applications. Science of Computer Programming, Vol. 68, No. 1, 21-37. (2007)
30. McPeak, S., Necula, G.C. Elkhound: A Fast, Practical GLR Parser Generator. International Conference on Compiler Construction (CC'04), Barcelona, Spain, Lecture Notes in Computer Science, Vol. 2985, 73-88. (2005)
31. Mernik, M., Lenic, M., Acdicausevic, E., Zumer, V.: LISA: An Interactive Environment for Programming Language Development. 11th International Conference on Compiler Construction (CC'02), Grenoble, France, Lecture Notes in Computer Science, Vol. 2304, Springer, 1-4. (2002)
32. Mössenböck, H. A Generator for Production Quality Compilers. 3rd intl. workshop on Compiler Compilers (CC'90), Schwerin, Lecture Notes in Computer Science Vol. 477, 42–55. (1990)
33. Oliveira, N., Varanda-Pereira, M.J., Henriques, P.R., da Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science and Information Systems Journal, Vol. 7, No. 2, 266-289. (2010)
34. Paakki, J. Prolog in Practical Compiler Writing. Computer Journal, Vol. 34, No. 1, 64-72. (1991)
35. Paakki, J.: Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, Vol. 27, No. 2, 196-255. (1995)
36. Paakki, J.: PROFIT: A System Integrating Logic Programming and Attribute Grammars. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'91), Passau, Germany, Lecture Notes in Computer Science Vol. 528, 243-254. (1991)
37. Parr, T., Fisher, K. LL(*): the Foundation of the ANTLR Parser Generator. ACM SIGPLAN Notices - PLDI '11, Vol. 46, No. 6, 425-436. (2011)
38. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. (2007)
39. Rebernak, D., Mernik, M., Henriques, P.R., Carneiro, D., Varanda-Pereira, M.J. Specifying Languages Using Aspect-oriented Approach: AspectLISA. Journal of Computing and Information Technology, Vol. 4, 343-350. (2006)
40. Rebernak, D., Mernik, M., Henriques, P.R., Varanda-Pereira, M.J.: AspectLISA: An Aspect-oriented Compiler Construction System Based on Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, Vol. 164, 37-53. (2006)

41. Rodriguez-Cerezo, D., Sarasa, A., Sierra, J.L.: Implementing Attribute Grammars Using Conventional Compiler Construction Tools. 3rd Workshop on Advances in Programming Languages (WAPL'11), Szczezin, Poland, IEEE, 855-862. (2011)
42. Saraiva, J., Swiestra, D.: Generic Attribute Grammars. 2nd Workshop on Attribute Grammars and Their Applications (WAGA'99), Amsterdam, The Netherlands. (1999)
43. Sarasa, A., Temprado-Battad, B., Sierra, J.L, Fernández-Valmayor, A.: XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services, Bradford, UK, Proceedings of AINA'09 Workshops, IEEE, 322-327. (2009)
44. Sassa, M., Ishizuka, H., Nakata, I. Rie, a compiler generator based on a one-pass-type attribute grammar. Software – Practice & Experience, Vol. 25, No. 3, 229-250, (1995)
45. Schreiner, A.T., Friedman, H.G. Introduction to Compiler Construction with Unix. Prentice-Hall. (1985)
46. Scott, E., Johnstone, A. Right nulled GLR parsers. ACM Transactions on Programming Languages and Systems, Vol. 28, No. 4, 577-618. (2006)
47. Sierra, J.L., Fernández-Valmayor, A. A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, Vol. 164, 19-36. (2006)
48. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher-Order Attribute Grammars. ACM SIGPLAN Notices Vol. 24, No. 7. (1989)
49. Vyk, E.R.v., Schwerdfeger, A.C. Context-aware scanning for Parsing Extensible Languages. 6th International Conference on Generative Programming and Component Engineering GPCE'06, Portland, Oregon, USA, ACM, 63-72. (2006)
50. Walsteijn, M.J., Kuiper, M.F.: Attribute Grammars in Prolog. Technical Report, RU-CS-86-14, Utrecht University. (1986)
51. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: An Extensible Attribute Grammar System. Science of Computer Programming, Vol. 75, No. 1-2, 39-54. (2010)

Daniel Rodríguez-Cerezo is a PhD student in the Computer Science School at UCM, and a member of the research group ILSA (Implementation of Language-Driven Software and Applications: http://ilsa.fdi.ucm.es). His research is focused on the use of several e-Learning techniques (simulations, interactive prototyping tools, recommendation systems for learning object repositories, etc.) to improve teaching and learning of the Software Language Engineering discipline. Besides, he is interested in the development and improvement of software language engineering techniques.

Antonio Sarasa-Cabezuelo is a full-time Lecturer in the Computer Science School at Complutense University of Madrid, Spain (UCM). His research is focused on the language-oriented development of XML-processing applications, and on the development of applications in the fields of digital humanities and e-Learning. He was one of the developers of the Agrega project on digital repositories (a pioneer project in this field in Spain). He is a member of ILSA. He has participated in several research projects in the fields

of software language engineering, digital humanities and e-learning, and he has published over 50 research papers in national and international conferences.

José-Luis Sierra is an Associate Professor at the UCM's Computer Science School, where he leads the ILSA Research Group. His research is focused on the development and practical uses of computer language description tools and on the language-oriented development of interactive and web applications in the fields of digital humanities and e-Learning. Prof. Sierra has leaded and participated in several research projects in the fields of digital humanities, e-learning and software language engineering, the results of which have been published in over 100 research papers in international journals, conferences and book chapters. He serves regularly as reviewer / PC Member for several international reputed journals and conferences.

Received: December 23, 2011 Accepted: June 1, 2012.

6.8 Implementing attribute grammars using conventional compiler construction tools

Cita completa:

Rodriguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. Implementing Attribute Grammars Using Conventional Compiler Construction Tools. En FedCSIS'11: Proceedings of Federated Conference on Computer Science, 855-862. 2011.

Resumen original de la contribución:

This article describes a straightforward and structure-preserving coding pattern to encode arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up parser generation tools. According to this pattern, a bottom-up oriented translation scheme is systematically derived from the original attribute grammar. Semantic actions attached to each syntax rule are written in terms of a small repertory of primitive attribution operations. By providing alternative implementations for these attribution operations, it is possible to plug in different semantic evaluation strategies in a seamlessly way (e.g., a demand-driven strategy, or a data-driven one). The pattern makes it possible the direct implementation of attribute grammar-based specifications using widely-used translation scheme-driven tools for the development of bottom-up language translators (e.g. YACC, BISON, CUP, etc.). As a consequence, this initial coding can be subsequently refined to yield final efficient implementations. Since these implementations still preserve the ability of being extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.

Referencia de citas bibliográficas:

[5][7][47][53][74][85][86][89][101][108][120]

Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 855–862

ISBN 978-83-60810-22-4

Implementing Attribute Grammars Using Conventional Compiler Construction Tools

Daniel Rodríguez-Cerezo Antonio Sarasa-Cabezuelo, José-Luis Sierra
Facultad de Informática. Universidad Complutense de Madrid. 28040 Madrid, Spain
Email: drodricerezo@gmail.com, {asarasa, jlsierra}@fdi.ucm.es

***Abstract*—This article describes a straightforward and structure-preserving coding pattern to encode arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up parser generation tools. According to this pattern, a bottom-up oriented translation scheme is systematically derived from the original attribute grammar. Semantic actions attached to each syntax rule are written in terms of a small repertory of primitive *attribution* operations. By providing alternative implementations for these attribution operations, it is possible to plug in different semantic evaluation strategies in a seamlessly way (e.g., a *demand-driven* strategy, or a *data-driven* one). The pattern makes it possible the direct implementation of attribute grammar-based specifications using widely-used translation scheme-driven tools for the development of bottom-up language translators (e.g. YACC, BISON, CUP, etc.). As a consequence, this initial coding can be subsequently refined to yield final efficient implementations. Since these implementations still preserve the ability of being extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.**

I. Introduction

ATTRIBUTE grammars, which were introduced by Donald E. Knuth [8] as an extension of context-free grammars for describing the syntax and semantics of context-free languages, are widely-used as a high-level specification method for the first stages of the design and implementation of a computer language [1][11].

In order to make an attribute grammar – based specification executable, it is possible to use one of the many specialized tools supporting the formalism (see, for instance, [3] [10][11]). However, regardless the realized advantages of these tools, in practice, traditional implementations of language processors are rarely based on artifacts directly generated from attribute grammars. On the contrary, attribute grammars are taken as initial specifications of the tasks to carry out, while final implementations are usually achieved by using scanner and parse generators (e.g., ANTLR, CUP, Flex, Bison…), general-purpose programming languages, or a suitable combination of both techniques [1]. The process of transforming the initial specification in a final implementation is usually ill-defined, and usually depends solely on the programmer's art, who many times discards formal specifications while directly hacks the final implementation. It seriously hinders systematic development and maintenance of language processors.

In order to bridge the gap between attribute grammar-based specifications and final implementations, we propose to articulate the language processor development process as the explicit transformation of the initial attribute grammar-based specification to the final implementation. According to our proposal, the first step to convey during the implementation stage is to explicitly encoding the attribute grammar in the input language of the development tool (usually, a parse generator like Bison or CUP). It will make it possible to yield an initial running implementation, which subsequently can be refined to achieve greater efficiency. In addition, since the refined implementation still supports the explicit incorporation and subsequent refinement of attribute grammar – based features, the incremental development and subsequent maintenance of the language processor can be largely facilitated.

This paper is focused on the first step of our proposal, i.e. how to code an attribute grammar in terms of the input language supported by a conventional parse generation tool. More precisely, we will focus on bottom-up parse generators of the YACC and CUP type. Unlike to works in LR-attributed grammars [2] and similar approaches (e.g., [6]), our approach will support the implementation of arbitrary non-circular attribute grammars. In addition, the encoding pattern will be independent of the final evaluation style chosen. Indeed, attribute grammars will be coded using a small repertory of *attribution* operations. Finally, by providing alternative implementations for these operations, it will be possible to set up the semantic evaluation style finally used.

The structure of the rest of the paper is as follows: section II describes the encoding pattern itself. Sections III and IV show how to plug in different evaluation styles by providing suitable implementations of the attribution operations. Finally, section V concludes the paper and outlines some lines of future work.

II. Encoding the attribute grammars

In this section we introduce our coding pattern. In order to make it as general as possible, we will not compromise with any particular generation tool, and we will use pseudo-code comprising very simple and standard procedural interfaces and imperative constructs. In addition, we will use a YACC-like notation [1] to refer to semantic values of symbols in the parse stack. In subsection II.A we describe the basic attribution operations allowed in the syntax rules' semantic actions. In sections II.B and II.C we describe the coding pattern itself, and in section II.D we exemplify it.

Table I.
Attribution operations

Operation	Intended Meaning
mkCtx(*n*)	It creates and initializes a list of *n* attribute instances for a symbol in the parse stack.
mkDep (a_0, a_1)	It sets a dependency between two attribute instances. Indeed, it declares the attribute instance a_0 depends on the attribute instance a_1.
inst(*a*,*f*)	It *instruments* the attribute instance *a* by establishing *f* as the semantic function to be applied during evaluation (*f* is actually an integer identifier of such a semantic function)
release(*as*)	It invokes garbage collection on the list of attribute instances *as*.
release(*a*)	It invokes garbage collection on the attribute instance *a*
set(*a*,*val*)	It fixes the value of the attribute instance *a* to *val*.
val(*a*)	It retrieves the value of the attribute instance *a*.

A. Attribution operations

Our coding pattern is largely based on the explicit description of the attribution structure of each grammar rule. For this purpose, we introduce the repertory of basic *attribution* operations outlined in Table 1. This table shows both the procedural interfaces of the operations and their intended meanings.

As such a description makes apparent, the purpose of these operations is to provide the developer with the necessary tools to describe how the *attribute dependency graph* associated with a sentence can be built conforming this sentence is analyzed by the parser. In addition, it also lets the developer indicate the semantic functions for computing each attribute instance. It does not necessarily means the graph must be fully stored in memory: depending on the actual implementation of the attribution operations, it will be possible to optimize, to a greater or lesser extent, the heap overhead (see sections III and IV).

B. Writing the translation scheme

The actual encoding of the attribute grammar requires writing a translation scheme describing how to build the aforementioned attribute dependency graph for each processed sentence. It can be done in a straightforward way by applying the following guidelines to each rule of the attribute grammar:

- First at all, we need to create the semantic value for the rule left-hand side (LHS). It is done by using an `mkCtx` operation. We only need to indicate the number of semantic attributes for the LHS.
- Next, we need to describe the dependencies between the attribute instances. Such dependencies are directly determined by examining the semantic equations, and they must be stated using the `mkDep` operation.
- Once it has been done, it is necessary to *instrument* the synthesized attribute instances in the rule's LHS, as well as the inherited attribute instances of the symbols in the rule's right-hand side (RHS). Once more, the code is straightforward: an `inst` operation for each equation. Notice we need to encode the semantic functions with integer identifiers, which can be interpreted by a *semantic function manager* (see subsection II.C).
- Finally, we need to release the attribute instance lists for the symbols in the rule's RHS.

Concerning the allocation of lexical attribute instances, it must be performed by the scanner, which will return the corresponding attribute instance list using a suitable field in the token. Also, notice the underlying context-free grammar must belong to the kind of grammars supported by the parser generation tool. Since we are using bottom-up parse generation translators, which usually support LALR(1) grammars [1], in practice it does not suppose a serious limitation.

C. Writing the Semantic Function Manager

In addition to the translation scheme, we need to code another auxiliary component, the *semantic function manager*, supporting the execution of the semantic functions. This component can be conceived as a procedure that, taking the semantic function's identifier and the sequence of attribute instances as input, returns the result of applying the function to the attribute instances.

D. Example

To illustrate the pattern we will consider the attribute grammar in Fig. 1. It models a very simple processor that makes it possible to evaluate simple arithmetic expressions involving addition and multiplication. To store the value we use a `val` synthesized attribute. Additionally, the processor can use a memory of predefined constants, which is propagated using an `env` inherited attribute.

Fig. 2 shows the translation scheme for this attribute grammar. In order to make the encoding more readable, we intro-

duce some constants for attribute instance indexes and for semantic function integer identifiers.

```
E ::= E + T
  E1.env↓ = E0.env↓
  T.env↓ = E0.env↓
  E0.val↑ = E1.val↑ + T.val↑
E ::= T
  T.env↓ = E.env↓
  E.val↑= T.val↑
T ::= T * F
  T1.env↓ = T0.env↓
  F.env↓ = T0.env↓
  T0.val↑ = T1.val↑ * F.val↑
T ::= F
  F.env↓ = T.env↓
  T.val↑ = F.val↑
F ::= n
  F.val↑ = toNum(n.lex↑)
F ::= id
  F.val↑ = valOf(F.env↓,id.lex↑)
F ::= ( E )
  E.env↓ = F.env↓
  F.val↑= E.val↑
```

Fig 1. Example attribute grammar. To improve readability, synthesized attribute occurrences are suffixed with ↑, and inherited occurrences are suffixed with ↓.

In this translation scheme, the first rule (which is not present in the original grammar) plays the role of initiating the processing. Indeed:

- It sets `env` in the root of the parse tree (we suppose the environment is returned by the `getEnv` external procedure).
- Then, it prints the value of `val` in such a root.
- Finally, it releases the root's attribute instance list.

The other rules are obtained by a step-by-step application of the guidelines described in the previous subsection. For instance, the encoding (shadowed in Fig. 2) of the first rule of the attribute grammar (shadowed in Fig. 1) is obtained as follows:

- Since `E`, the rule's LHS, has two semantic attributes (`env` and `val`), we need to invoke `mkCtx` with 2 as the number of attributes to be allocated. Notice that the resulting attribute instance list is assigned to `$$`, which in YACC-like notation is the pseudo-variable for the semantic value of the rule's LHS.
- From the first equation, we get E_1`.env` depends on E_0`.env`. This dependency is declared by `mkDep($1[env],$$[env])`, since (i) `$1` refers, in YACC-like notation, to the semantic value of E_1, and (ii) `$$` refers, as said before, to the semantic value of E_0.
- In a similar way, the other three `mkDep` actions are derived from the other two equations. Notice that the third equation yields two `mkDep` actions, since, according to it, E_0`.val` depends on two different attributes: E_1`.val` and `T.val`.
- In their turn, each equation yields an `inst` action. For doing so, firstly we need to identify the semantic function used in the equation. It can require some intermediate analysis. For instance, to make the semantic function apparent, E_1`.env`↓ = E_0`.env`↓ must be actually read as E_1`.env`↓ = $\lambda v(v) E_0$`.env`↓. Thus, we can assign an integer code to this $\lambda v(v)$ semantic function (in Fig. 2, this code is given by the IDEN constant). A similar technique can be used for equations sides involving more complex expressions. For instance, E_0`.val`↑ = E_1`.val`↑ + `T.val`↑ can be actually read as E_0`.val`↑ = $\lambda v_0(\lambda v_1(v_0+v_1))E_1$`.val`↑ `T.val`↑, which leads to identify $\lambda v_0(\lambda v_1(v_0+v_1))$ as the semantic function (it is identified by the ADD constant in Fig. 2).

```
def env=0; def val=1;
def IDEN=0; def ADD=1; def MUL=2;
def TONUM=3; def VALOF=4;
S ::= E {
  set($1[env],getEnv());
  print(val($1[val]));
  release($1);}
E ::= E + T {
  $$ := mkCtx(2);
  mkDep($1[env],$$[env]); mkDep($3[env],$$[env]);
  mkDep($$[val],$1[val]); mkDep($$[val],$3[val]);
  inst($1[env],IDEN);
  inst($3[env],IDEN);
  inst($$[val],ADD);
  release($1); release($3);}
E ::= T {
  $$ := mkCtx(2);
  mkDep($1[env],$$[env]); mkDep($$[val],$1[val]);
  inst($1[env],IDEN);
  inst($$[val],IDEN);
  release($1);}
T ::= T * F {
  $$ := mkCtx(2);
  mkDep($1[env],$$[env]); mkDep($3[env],$$[env]);
  mkDep($$[val],$1[val]); mkDep($$[val],$3[val]);
  inst($1[env],IDEN);
  inst($3[env],IDEN);
  inst($$[val],MUL);
  release($1); release($3);}
T ::= F {
  $$ := mkCtx(2);
  mkDep($1[env],$$[env]); mkDep($$[val],$1[val]);
  inst($1[env],IDEN);
  inst($$[val],IDEN);
  release($1);}
F ::= n {
  $$ := mkCtx(2);
  mkDep($$[val],$1[lex]);
  inst($$[val],TONUM);
  release($1);}
F ::= id {
  $$ := mkCtx(2);
  mkDep($$[val],$$[env]); mkDep($$[val],$1[lex]);
  inst($$[val],VALOF);
  release($1);}
F ::= ( E ) {
   $$ := mkCtx(2);
   mkDep($$[val],$2[val]);
   mkDep($2[env],$$[env]);
   inst($2[env], IDEN);
   inst($$[val],IDEN);
   release($2);}
```

Fig 2. Encoding of the attribute grammar in Fig. 1.

- Finally, we include a `release` action for each symbol in the rule's RHS having semantic attributes.

Finally, in addition to the translation scheme, we need to provide a suitable semantic function manager. It is depicted by the pseudo-code in Fig. 3. Basically, it is a dispatcher that, according to the function's integer identifier, applies the actual function on the sequence of semantic attribute instances[1] .

```
procedure exec(FUN,ARGS) {
case FUN of
  IDEN →
   return val(ARGS[0]);
  ADD →
   return val(ARGS[0]) + val(ARGS[1]);
  MUL →
   return val(ARGS[0]) * val(ARGS[1]);
  TONUM →
   return toNum(val(ARGS[0]));
  VALOF →
   return valOf(val(ARGS[0]),val(ARGS[1]));
 end case
}
```

Fig 3. Semantic function manager for the attribute grammar in Fig. 1.

III. Incorporating a demand-driven evaluation framework

In order to make possible the execution of the encodings proposed in the previous section, we need to implement the basic attribution operations. In this section we describe a straightforward implementation supporting a *demand-driven* evaluation style (see, for instance [5] [9]). In this implementation, semantic evaluation starts once the sentence has been completely parsed. In this point, there is an in-memory representation of the part of the dependency graph required for performing semantic evaluation. During evaluation, the values of the attribute instances will be calculated only when they are required.

In the following subsections we describe the resulting framework explaining how attribute instances are represented (subsection III.A). Then, we specify how the attribution operations work (subsection III.B). Finally, we illustrate the complete framework with an example (subsection III.C). For the sake of simplicity, we will ignore the detection of potential circularities in the underlying dependency graphs, although it would not be difficult to extend the framework to support it.

A. Representing the instances of the semantic attributes

The instances of the semantic attributes can be conceived as records. Table 2 outlines the fields required together with their intended purposes. Thus, this representation makes it possible to build a dependency structure in which:

- Each attribute instance points to those attribute instances required to compute it (in a similar way to the *reversed* dependency graph used in [5]).
- In addition, it explicitly stores the identifier of the semantic function to be used in this computation.

[1]Remark that, by the sake of generality, we are intentionally using a minimal set of programming language features. Indeed, by using a more sophisticated programming paradigm (e.g., a language equipped with higher-order features), it could be possible to give more elegant solutions to these basic conceptualizations.

Table II.
Structure of attribute instances in the demand-driven evaluation framework.

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	$\perp$
available	A boolean flag that indicates whether the value is available.	false
deps	It keeps the links to those attribute instances required to compute the value.	The *empty list*
semFun	It stores the integer code of the semantic function required to compute the value.	$\perp$
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

B. Implementing the attribution operations

Table 3 outlines, using pseudo-code, the implementation of the attribution operations. In this pseudo-code, references are intended to work like in Java, although we do not assume automatic garbage collection (instead, a `delete` primitive is explicitly invoked). Indeed, this is why we explicitly include `release` attribution operations.

The different operations behave as follows:

- `mkCtx` collects in a list many fresh attribute instances as needed.
- `mkDep` adds the second attribute instance in the `deps` list of the first one.
- `inst` stores the semantic function code in the `semFun` field.
- `release`, when applied to a list of semantic attribute instances, releases each instance and de-allocates the list itself.
- On the other hand, when `release` is applied to an attribute instance, decreases in 1 its reference count. If this count becomes 0, the instances on which it depends are released; finally, the original instance itself is de-allocated.
- `set` sets the `value` field and records its availability.
- `val` recovers the value of an attribute instance as follows: (i) if the value is available, it returns such a value, (ii) otherwise, it calls the semantic function manager to compute such a value and sets and returns it.

Thus, the demand-driven evaluation process arises from the interplay of the `val` attribution operation and the semantic function manager. Notice that, in our minimalistic conceptualization, we assume this manager has the pre-estab-

lished `exec` name, and its implementation is changed from encoding to encoding[2].

TABLE III.
IMPLEMENTATION OF THE ATTRIBUTION OPERATIONS FOR ALLOWING A DEMAND-DRIVEN EVALUATION STYLE.

Operation	Implementation

mkCtx(*n*)	*as* := **new list**
	for i := 0 **to** *n*-1 **do**
	  add(*as,* **new attribute**)
	end for
	return *as*

mkDep (a_0, a_1)	**add** (a_0.deps, a_1) a_1.refcount := a_1.refcount + 1
inst(*a*,*f*)	*a*.semFun := *f*
release(*as*)	**foreach** *a* **in** *as* **do** release(*a*) **end foreach** **delete** *as*

release(*a*)	*a*.refcount:= *a*.refcount-1
	if *a*.refcount = 0 **then**
	  foreach *a'* **in** *a*.deps **do**
	  release(*a'*)
	  end foreach
	  delete *a*.deps
	end if
	delete *a*

set(*a*,*val*)	*a*.value := *val* *a*.available := **true**

val(*a*)	**if** ¬ *a*.available **then**
	  set(*a*, exec(*a*.semFun,*a*.deps))
	  release(*a*.deps)
	end if
	return *a*.value

Also notice how explicit garbage collection can be readily interleaved in the implementation of the attribution operation by appropriately managing the reference counters and by de-allocating lists and records as soon as they become unreachable. Although in this evaluation style, most of the dependency graph remains in memory until parsing is finished, automatic garbage collection makes it possible to de-allocate useless parts of the graph when they becomes unreachable. It can be due to attribute instances that are not finally required in any computation (e.g., `F.env` in a `F ::= id` context), or to successive evolutions of the implementation combining pure attribute grammar features with implementation-oriented optimizations (e.g., global variables, on-the-fly evaluation of semantic attributes, …).

C. Example

In order to illustrate the internal functioning of the framework, we will use the example developed in subsection II.D, together with the input sentence `5 * (6 + x)`.

[2]Again it is possible to achieve more elegant solutions by using a programming language with a minimal higher-order support (e.g., a conventional object-oriented language). Nevertheless, our conceptualization keeps the essence of this evaluation approach.

Action	Input	Parse Stack
1.init	*(6+x)$	
2. shift	(6+x)$	$n^{[1]}$
3. reduce F ::= n	(6+x)$	$F^{[2,3]}$
4. reduce T ::= F	(6+x)$	$T^{[4,5]}$
5. shift	6+x)$	$T^{[4,5]}*$
6. shift	+x)$	$T^{[4,5]}*($
7. shift	x)$	$T^{[4,5]}*(n^{[6]}$
8. reduce F ::= n	x)$	$T^{[4,5]}*(F^{[7,8]}$
9. reduce T ::= F	x)$	$T^{[4,5]}*(T^{[9,10]}$
10. reduce E ::= T	x)$	$T^{[4,5]}*(E^{[11,12]}$
11. shift	)$	$T^{[4,5]}*(E^{[11,12]}+$
12. shift	$	$T^{[4,5]}*(E^{[11,12]}+id^{[13]}$
13. reduce F ::= id	$	$T^{[4,5]}*(E^{[11,12]}+F^{[14,15]}$
14. reduce T ::= F	$	$T^{[4,5]}*(E^{[11,12]}+T^{[16,17]}$
15. reduce E ::= E+T	$	$T^{[4,5]}*(E^{[18,19]}$
16. shift	$	$T^{[4,5]}*(E^{[18,19]})$
17. reduce F := (E)	$	$T^{[4,5]}*F^{[20,21]}$
18. reduce T := T*F	$	$T^{[22,23]}$
19. reduce E := T	$	$E^{[24,25]}$
20. reduce S := E	$	S

Fig 4. Evolution of the translator generated from Figure 2 while analyzing `5*(6+x)`

Fig. 4 illustrates the evolution of the parser. Each symbol in the parse stack is superscripted by the list of the references to their attribute instances. Fig. 5 outlines the dependency structure created in the heap. In this structure, nodes correspond to attribute instances, while dependencies are indicated by mean of arrows. Each instance is accompanied by a numeric identifier (it is also used to indicate references in the parse stack), and by its *duration* (i.e., the parse action in which the instance was created, followed by the parse action in which it was deleted). For example, the instance 16 was created in the action 14 (reduction of the `T ::= F` rule), and it was destroyed in action 20 (once the parsing concluded and semantic evaluation was activated as a consequence of consulting `val` in the parse tree root).

Finally, it is important to remark several points. On one hand, it should be notice the parse tree is never explicitly built, since the process only requires the underlying dependency graph. Additionally, dependencies in Fig. 5 are reverted with respect to the usual convention, according to which, when *a* is used to compute *b*, an arc starting in *a* and finishing in *b* is used [8]. Indeed, dependencies actually represent the contents of the `deps` field. Additionally, the fact they appear reversed with respect to the usual convention em-

phasized the demand-driven nature of this evaluation strategy. Also, while most of the instances exists until the end of the process, this example makes apparent how those instances that become unreachable are readily de-allocated. For example, consider action 4: when rule `T ::= F` is reduced, the instance for `F.env` (i.e., the instance 3) is no longer needed, and therefore it can be de-allocated. Actually, this instance is de-allocated as consequence of releasing the `F` attribute instance list. By last, notice how lexical attributes for terminal symbols are created in the *shift* action that precedes the actual shift of such a symbol, or during parse initialization, in the case of the first shift. It is due to lexical attributes are actually created by the scanner, and we suppose 1-lookahead parsers, as those generated by LALR(1) parser generators.

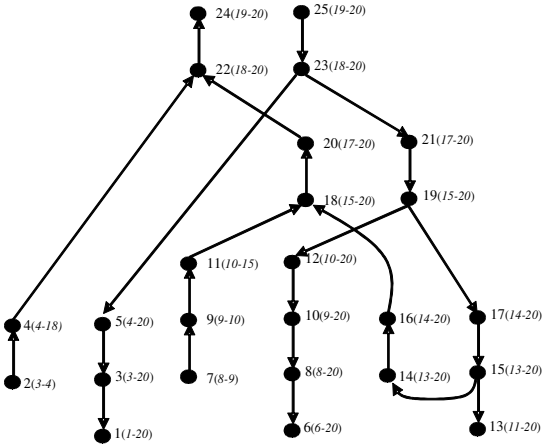


Fig 5. Dependency structure created in the heap as by the process outlined in Fig. 4.

IV. Incorporating a data-driven evaluation framework

In this section we describe an alternative implementation of the attribution operations, which leads to a *data-driven* evaluation style (see, for instance, [7]). In this evaluation style, attribute instances are scheduled for being evaluated as soon as the values for all the instances on which it depends are available. Thus, this method can shorten the durations of attribute instances. Additionally, it can interleave evaluation with parsing. These features can result of interest to process very long sentences, or sentences made available asynchronously (e.g., on a network communication channel). However, this method can do useless evaluations on attribute instances not required to yield the final results.

As in the previous section, we outline the representation of attribute instances (subsection IV.A), the implementation of attribute operations (subsection IV.B), and we illustrate how the method works with an example (subsection IV.C).

A. Representing the instances of the semantic attributes

Table 4 outlines the representation of attribute instances in the data-driven style. Notice that, in addition to the list of instances on which an instance depends, it is needed to maintain the reverse relationship (i.e., each attribute instance must refer to those instances which depend on it). Indeed, this representation is similar to the used by networks of *observables-observers* in the *observer* object-oriented pattern [4][3].

Table IV.
Structure of attribute instances in the data-driven evaluation framework.

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	⊥
available	A boolean flag that indicates whether the value is available.	false
deps	It keeps the links to those attribute instances required to compute the value.	The *empty list*
obs	It keeps the links to those attribute instances *observing* it (i.e., which depend on it to compute their values).	The *empty list*
required	Counter which records the number of attribute instances in *deps* whose values have not yet been determined.	0
semFun	It stores the integer code of the semantic function required to compute the value.	⊥
instrumented	*True* if *semFun* was set, *false* otherwise.	false
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

B. Implementing the attribution operations

Table 5 outlines the pseudo-code of the attribution operations whose implementation differs from those in the demand-driven style. In this way, we only need to redefine `mkDep`, `inst`, `set` and `val`:

- In addition to updating `deps` in the first instance, `mkDep` must test whether the second instance was already computed. If it is not available, the first instance must be added to its `obs` list, since such an instance depends on its value, a value which is not yet available.
- On its hand, `inst` must take care of whether the value can be computed. Indeed, if the corresponding attribute instance has all the instances on which it depends computed, it can thereby be computed. It assumes the establishment of all the required dependencies before instrumentation, which is ensured by our encoding pattern.
- `Set` must take care of decrementing the `required` counters in all the instances depending

[3]As with the demand-driven style, this representation could be simplified, inferring the values of flags (in this case, *available* and *instrumented*) from the other fields. However, we prefer to explicitly preserve these flags to increase the readability of pseudo-code.

of the current one. In addition, if a counter becomes 0, it must enforce the evaluation of the corresponding instance.

- Finally, `val` immediately computes the value, unless the instance has not been yet instrumented.

TABLE V.
IMPLEMENTATION OF THE ATTRIBUTION OPERATIONS FOR ALLOWING A DATA-DRIVEN EVALUATION STYLE (ONLY THOSE IMPLEMENTATIONS DIFFERING FROM TABLE III ARE PRESENTED).

Operation	Implementation

mkDep (a_0, a_1)	**add** (a_0.deps, a_1)
	a_1.refcount := a_1.refcount + 1
	if ¬ a_1.available **then**
	  add (a_1.obs, a_0)
	  a_0.required := a_0.required + 1
	  a_0.refcount := a_0.refcount + 1
	end if

inst(*a*,*f*)	*a*.semFun := *f*
	a.instrumented := **true**
	if *a*.required = 0 **then**
	  val(*a*)
	end if

set(*a*,*val*)	*a*.value := *val*
	a.available := **true**
	foreach *a*' **in** *a*.obs **do**
	  a'.required := *a*'.required – 1
	  if *a*'.required = 0 **then**
	  val(*a*')
	  end if
	end foreach
	release(*a*.obs)

val(*a*)	**if** *a*.instrumented **then**
	  set(*a*, exec(*a*.semFun,*a*.deps))
	  a.available := **true**
	  release(*a*.deps)
	end if

Notice how, in this case, evaluation can be interleaved with parsing. Indeed, evaluation is fired when the values of attribute instances are explicitly set, and also when attributes are instrumented. As a consequence, garbage collection also interplays with parsing, and, therefore, this method can incur in less heap overhead. It can be realized by considering the implementation of an *s-attributed* grammar (i.e., a grammar with only synthesized attributes) [1]. In this case, LHS attribute instances are computed when they are instrumented, and RHS attribute instances are garbage collected immediately before the reduction of the corresponding rules. On other cases, the behavior strongly depends on the nature of inherited attributes. In the extreme case (e.g., the example developed in subsection II.D), the dependency structure will be fully constructed, and evaluation will be delayed until the end of parsing, as in the demand-driven style. Still in these cases, it is possible to apply some straightforward optimizations on the resulting encoding, based on the use of *marker* non-terminals [1], in order to improve performance.

```
def env=0; def val=0;
def IDEN=0; def ADD=1; def MUL=2;
def TONUM=3; def VALOF=4;
S ::= M0 E {
  print(val($2[val]));
  release($1); release($2);}
M0 ::= λ {
  $$ := mkCtx(1);
  set($$[env],getEnv());}
E ::= E + M1 T {
  $$ := mkCtx(1);
  mkDep($$[val],$1[val]); mkDep($$[val],$4[val]);
  inst($$[val],ADD);
  release($1); release($3); release($4);}
M1 ::= λ {
  $$ := mkCtx(1);
  mkDep($$[env], $-2[env]);
  inst($$[env], IDEN) }
E ::= T {
  $$ := mkCtx(1);
  mkDep($$[val],$1[val]);
  inst($$[val],IDEN);
  release($1); }
T ::= T * M1 F {
  $$ := mkCtx(1);
  mkDep($$[val],$1[val]); mkDep($$[val],$4[val]);
  inst($$[val],MUL);
  release($1); release($3); release($4); }
T ::= F {
  $$ := mkCtx(1);
  mkDep($$[val],$1[val]);
  inst($$[val],IDEN);
  release($1);  }
F ::= n {
  $$ := mkCtx(1);
  mkDep($$[val],$1[lex]);
  inst($$[val],TONUM);
  release($1); }
F ::= id {
  $$ := mkCtx(1);
  mkDep($$[val],$0[env]); mkDep($$[val],$1[lex]);
  inst($$[val],VALOF);
  release($1); }
F ::= ( M2 E ) {
   $$ := mkCtx(1);
   mkDep($$[val],$3[val]);
   inst($$[val],IDEN);
   release($2); release($3);}
M2 ::= λ {
  $$ := mkCtx(1);
  mkDep($$[env], $-1[env]);
  inst($$[env], IDEN) }
```

Fig 6. Result of optimizing the translation scheme of Figure 2 with the help of markers to get the most of the data-driven evaluation style.

C. Example

In order to illustrate the potential advantages of the data-driven method with respect to heap requirements, we will slightly modify the encoding of Fig. 2 by introducing marker non-terminals (i.e., new non-terminal symbols defined by empty rules) in strategic places[4].

These marker non-terminals will allocate references to the `env` attribute instance for their immediate successors in the parse stack. It lets us discard equations to propagate the environment along the parse tree left-spines. The resulting encoding is shown in Fig. 6.

Fig. 7 illustrate the evolution of the parser while analyzing the sentence `5 * (6 + x)`. Fig. 8 shows the dependency structure created in the heap. Notice how the marker-based optimization performed on the encoding makes it possible to get a behavior equivalent to a *one-pass*, *on-the-fly*, evaluation of the semantic attributes, as the durations indicated in Fig. 8 make apparent.

[4]It must be carefully done, as the resulting context-free grammar can lose its desired character -e.g., LALR(1).

Action	Input	Parse Stack
1.init	*(6+x)$	
2. reduce **M0** ::= λ	*(6+x)$	**M0**[2]
3. shift	(6+x)$	**M0**[2]n[1]
4. reduce F ::= n	(6+x)$	**M0**[2]F[3]
5. reduce T ::= F	(6+x)$	**M0**[2]T[4]
6. shift	6+x)$	**M0**[2]T[4] *
7. reduce **M1** ::= λ	6+x)$	**M0**[2]T[4]* **M1**[5]
8. shift	+x)$	**M0**[2]T[4]***M1**[5] (
9. reduce **M2** ::= λ	+x)$	**M0**[2]T[4]* **M1**[5] (**M2**[7]
10. shift	x)$	**M0**[2]T[4]* **M1**[5] (**M2**[7] n[6]
11. reduce F ::= n	x)$	**M0**[2]T[4] ***M1**[5] (**M2**[7] F[8]
12. reduce T ::= F	x)$	**M0**[2]T[4] ***M1**[5] (**M2**[7] T[9]
13. reduce E ::= T	x)$	**M0**[2]T[4] ***M1**[5] (**M2**[7] E[10]
14. shift	)$	**M0**[2]T[4] ***M1**[5] (**M2**[7] E[10] +
15. reduce **M1** ::= λ	)$	**M0**[2]T[4] ***M1**[5] (**M2**[7] E[10] + **M1**[12]
16. shift	$	**M0**[2]T[4] ***M1**[5] (**M2**[7] E[10] + **M1**[12]id[11]
17. reduce F ::= id	$	**M0**[2]T[4] ***M1**[5] (**M2**[7] E[10] + **M1**[12]F[13]
18. reduce T ::= F	$	**M0**[2]T[4] ***M1**[5] (**M2**[7]E[10] + **M1**[12]T[14]
19. reduce E ::= E+**M1**T	$	**M0**[2]T[4] ***M1**[5] (**M2**[7]E[15]
20. shift	$	**M0**[2]T[4] ***M1**[5] (**M2**[7]E[15])
21. reduce F := (**M2**E)	$	**M0**[2]T[4] ***M1**[5]F[16]
22. reduce T := T***M1**F	$	**M0**[2]T[17]
23. reduce E := T	$	**M0**[2]E[18]
24. reduce S := **M0** E	$	S

Fig 7. Evolution of the translator generated from Figure 6 while analyzing `5*(6+x)`

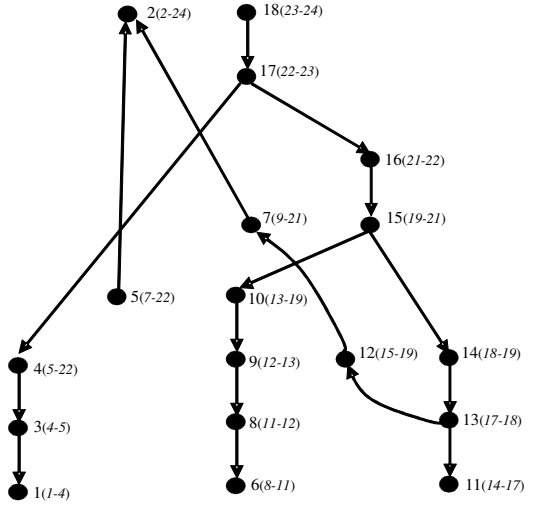


Fig 8. Dependency structure created in the heap as by the process outlined in Fig. 7

V. Conclussions and future work

This paper has shown how to systematically encode arbitrary non-circular attribute grammars in the input languages of bottom-up, LALR(1) parse generation tools like YACC, BISON or CUP. It is done using a small set of attribution operations. These operations, in their turn, can be implemented of different ways in order to enable different semantic evaluation styles. In particular, this paper has illustrated two alternative implementations: one supporting a demand-driven style, and another one supporting a data-driven one. The results of this work can be useful to promote a systematic method of using conventional bottom-up parse generation tools to yield final implementations. This method starts with the initial encoding of an attribute grammar-based specification, and then it evolves it in a final implementation by applying systematic implementation patterns and techniques. Besides, the method facilitates the incremental introduction of new language features, since they can be described according to attribute grammar conventions, then readily encoded in the implementation, and finally optimized according to implementation-dependent criteria. Therefore, the method transports the attribute grammar amenability for doing modular and extensible specifications incrementally to an implementation process based on parse generation tools.

Currently we have successfully tested our method with several small examples, and we are applying it to the development of a non-trivial translator for a Pascal-like language. As future work, we plan to apply the method to descent parser generation tools (e.g., JavaCC or ANTLR).

Acknowledgment

Thanks are due to the project grants TIN2010-21288-C02-01.

References

[1] Aho A.V, Lam M.S, Sethi R, Ullman J.D. 2006. Compilers: principles, techniques and tools (2nd Edition). Addison-Wesley.
[2] Akker, R; Melichar, B.; Tarhio, J. The Hierarchy of LR-attributed grammars. WAGA'90, Paris, France, September 19-21. 1990
[3] Ekman, T., Hedin, G. The JastAdd system - modular extensible compiler construction. Sc. of Comp. Prog. 69(1-3), 14-26. 2007
[4] Gamma,E ; Helm,R ; Jhonson,R ; Vlissides,J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995
[5] Jalili, F. A general linear-time evaluator for attribute grammars. ACM SIGPLAN Notices 18(9), 35-44. 1983
[6] Katwijk, J. A preprocessor for YACC or a poor man's approach to parsing attributed grammar. ACM SIGPLAN Notices 18(10), 12-15. 1983
[7] Kennedy, K.; Ramanathan, J. A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing. ACM Transaction of Programming Languages and Systems 1(1), 142-160. 1979
[8] Knuth, D. E. Semantics of Context-free Languages. Math. System Theory 2(2), 127–145. 1968. See also the correction published in Math. System Theory 5, 1, 95–96
[9] Magnusson, E.; Hedin, G. Circular reference attributed grammars—their evaluation and applications. Sc. of Comp. Prog. 68(1), 21-37. 2007
[10] Mernik, M.,Lenic, M., Acdicausevic, E., Zumer, V. LISA: An Interactive Environment for Programming Language Development. CC 2002, Grenoble, France, April 8-12, 2002
[11] Paaki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Comp. Surveys, 27, 2, 196-255. 1995

6.9 Serious games in tertiary education: A case study concerning the comprehension of basic concepts in computer language implementation courses

Cita completa:

Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J. L. Serious Games in Tertiary Education: A Case Study Concerning the Comprehension of Basic Concepts in Computer Language Implementation Courses. Computers in Human Behavior, 31: 558-570. 2014.

Resumen original de la contribución:

This paper describes Evaluators, a system for the development of educational serious games oriented to introductory computer language implementation courses similar to those included in Computer Science tertiary curricula. Evaluators lets instructors generate games from collections of exercises addressing basic concepts about the design and implementation of computer languages (in particular, the processing of artificial languages according to the model of attribute grammars). By playing the generated games, students interactively learn the fundamentals of the semantic evaluation process behind attribute grammars. Indeed, they implicitly find solutions to the exercises presented, and they receive immediate feedback about successful and incorrect actions. In addition, the games log students' actions, which can subsequently be analyzed by the instructors using a specialized analytic tool that is included in Evaluators. Assessment of the system, which was performed according to three different dimensions (the instructors' perspective, the students' perspective and educational effectiveness perspective), (a) indicates that the exercise-driven approach of Evaluators is a cost-effective approach amenable to extrapolation to other areas of Computer Science tertiary education, (b) shows a positive attitude of students toward the serious games built with Evaluators, and (c) evidences a positive effect of the system and its pedagogical strategy on long-term student performance.

Referencia de citas bibliográficas:

[2][5][6][10][13][19][20][24][32][38][40][45][54][57][58][65][64][75][76][77]
[81][89][94][95][109][110][116][117][120][124][125][131][132][133][137]
[142][145][156][157][158][165][168][174][177][178][179][181]

Computers in Human Behavior 31 (2014) 558–570

Contents lists available at SciVerse ScienceDirect

Computers in Human Behavior

journal homepage: www.elsevier.com/locate/comphumbeh

Serious games in tertiary education: A case study concerning the comprehension of basic concepts in computer language implementation courses

Daniel Rodríguez-Cerezo, Antonio Sarasa-Cabezuelo, Mercedes Gómez-Albarrán, José-Luis Sierra *

Dpto. Ingeniería del Software e Inteligencia Artificial, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain

ARTICLE INFO

Article history:
Available online 11 July 2013

Keywords:
Serious game
Computer Science education
Authoring tool
Learning analytics
Attribute grammar

ABSTRACT

This paper describes *Evaluators*, a system for the development of educational serious games oriented to introductory computer language implementation courses similar to those included in Computer Science tertiary curricula. *Evaluators* lets instructors generate games from collections of exercises addressing basic concepts about the design and implementation of computer languages (in particular, the processing of artificial languages according to the model of attribute grammars). By playing the generated games, students interactively learn the fundamentals of the semantic evaluation process behind attribute grammars. Indeed, they implicitly find solutions to the exercises presented, and they receive immediate feedback about successful and incorrect actions. In addition, the games log students' actions, which can subsequently be analyzed by the instructors using a specialized analytic tool that is included in *Evaluators*. Assessment of the system, which was performed according to three different dimensions (the instructors' perspective, the students' perspective and educational effectiveness perspective), (a) indicates that the exercise-driven approach of *Evaluators* is a cost-effective approach amenable to extrapolation to other areas of Computer Science tertiary education, (b) shows a positive attitude of students toward the serious games built with *Evaluators*, and (c) evidences a positive effect of the system and its pedagogical strategy on long-term student performance.

1. Introduction

Computer language implementation is a subject included in mainstream Computer Science curricular recommendations as a key aspect of a computer scientist's basic education (ACM/IEEE, 2008). Students usually consider it a difficult subject due to its intrinsically abstract nature, which in turn results in a lack of motivation in students compared to more practical topics in the curricula, such as programming technologies (Waite, Jarrahian, Jackson, & Diwan, 2006).

To facilitate the learning process of the students enrolled in a Compiler Construction course at the Complutense University of Madrid (Spain), we decided to use syntax-directed translation (Aho, Lam, Sethi, & Ullman, 2007) as a main paradigm for architecting language processors and *attribute grammars* (Paakki, 1995) as a basic formalism for the specification tasks. Our aim was to make students aware of the importance of clearly separating the specification of the different aspects of a language processor from its subsequent implementation. Based on our experience, the success of this teaching strategy depends heavily on the success that we have in teaching the basic concepts of the formalisms of attribute grammars and their underlying computational model during the early stages of the course. To help our students assimilate the attribute grammar essentials, in the 2009–2010 edition of the course, we created and provided students with batteries of exercises related to the comprehension level of Bloom's taxonomy (Bloom, Engelhart, Furst, Hill, & Krathwohl, 1956). Students completed exercises, each consisting of the following components:

- An informal description of a language processing task. This is a free-text description of the language processing problem to be addressed using attribute grammars.
- A formalization of the task by means of an attribute grammar. This is a formal specification using the attribute grammars' formalisms of the processing problem stated in the informal description.
- A sentence in the processed language (e.g., a program or a program fragment if the processed language is a programming language, which is the type of language commonly used in introductory Compiler Construction courses).

* Corresponding author. Tel.: +34 913947548; fax: +34 913947547.
E-mail address: jlsierra@fdi.ucm.es (J.-L. Sierra).

0747-5632/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.chb.2013.06.009

- The parse tree of this sentence, along with the semantic attribute instances whose values should be determined as a result of processing the sentence (e.g., the type of expression or the code generated for this expression if a programming language is considered).

To complete the exercises, students must enumerate an evaluation order for every semantic attribute to explain how the values of these attributes could be calculated in the attributed parse tree provided.

The learning reinforcement provided by solving the exercises proved satisfactory for helping students assimilate fundamental concepts of attribute grammars. For instance, the exercises helped them understand the dependency-driven computation strategy, according to which the evaluation order does not matter as long as the dependencies among attributes are preserved, similarly, for instance, to what happens in a spreadsheet. Unfortunately, however, we also observed that the lack of motivation remained. Indeed, many students found it boring to solve these exercises using pencil and paper. As a consequence, many of them provided incorrect or unfinished solutions, as the students' general tendency was to extrapolate the instructors' solution templates to the new exercises; the predominance of rote learning was alarming. Therefore, we considered creating more effective methods for overcoming the deficiencies detected.

Taking into account that serious games are a growing trend in the field of e-learning, as well as the significant success of this type of environment in recent years (Amory et al.,1999; de Freitas & Liarokapis, 2011; Gee, 2003; Jiménez-Díaz, Gómez-Albarrán, & González-Calero, 2007; Minua, Andreas, & Lakhmi, 2011), we decided to develop *Evaluators*, an educational system that lets instructors easily generate serious games following an exercise-driven approach. The serious games in *Evaluators* are generated from batteries of exercises concerning basic concepts of attribute grammars of the type described above. These exercises are developed using the authoring tool provided in *Evaluators*. The games generated involve students in interactive simulations of the semantic evaluation processes behind attribute grammars. Students determine the correct evaluation order for semantic attributes, and they receive immediate feedback on both successful and incorrect actions. Student actions are logged and can later be analyzed by the instructors using a specialized analytic tool included in *Evaluators*. *Evaluators* benefits from our experience in the development of educational simulations and games (Jiménez-Díaz, González-Calero, & Gómez-Albarrán, 2012; Jiménez-Díaz et al., 2007) and is an evolution of our previous work in the development of specialized tools related to the learning of language processing (Sierra, Fernández-Pampillón, & Fernández-Valmayor, 2008), in which students had to provide solutions to processing problems that were similar to the ones used in *Evaluators* but focused on writing specifications instead of comprehending them.

The structure of the rest of the paper is as follows: Section 2 presents several works that are related to this project; Section 3 provides an in-depth description of *Evaluators*; Section 4 reports some results of our assessment of the system; and Section 5 presents the conclusions and future work.

2. Related work

In this section, we summarize some works related to ours: pedagogical approaches followed in Compiler Construction courses (Section 2.1), visualization and simulation tools used in Computer Science education (Section 2.2), educational tools for enhancing the teaching and learning of concepts related to Compiler Construction (Section 2.3), and game-based teaching and learning of Computer Science topics (Section 2.4).

2.1. Teaching of Compiler Construction courses

There are different strategies for facilitating the assimilation of concepts in Compiler Construction courses. These strategies can be roughly grouped in the following categories:

- *Implementation of a processor for a small programming language*. This is the approach that is most widely used by instructors. In this approach, instructors describe a small programming language with a couple of basic data types, basic operations for these data types, control structures (such as loops and conditionals) and an abstraction mechanism. The students have to implement a compiler for the programming language. There are different prototypical languages used for this strategy, such as COOL (Aiken, 1996), MINIML (Baldwin, 2003) and CHIRP (Xu & Fred, 2006). Other less conventional proposals also exist, such as those in which students use languages for graph representation (Werner, 2003), simple figure drawing (Ruckert, 2007) or robot action programming (Xu & Fred, 2006). The main advantage of these languages is the motivational component that they possess.
- *Small language processing projects*. This strategy aims to solve two major problems based on the implementation of a whole language processor: the tangible possibility that students get stuck in the first phases of development and cannot accomplish the project on time and the need for mastering concepts for developing the language processor that are not taught until later in the course. For this purpose, using small projects focuses students on the concepts covered in class. Thus, as the course progresses, students are able to immediately apply the concepts studied in class. The works by Ledgard (1971) and Shapiro and Mickunas (1976) provide an in-depth discussion of this strategy.
- *Analysis and debugging of processors for real programming languages*. The main idea behind this strategy is that the implementation of any language compiler (a small one or small languages projects) is not sufficient to teach all the concepts involved in the development of real programming language processors. Thus, the work of White, Sen, and Stewart (2005) proposes teaching language processor internals by debugging the source code of real-world compilers. The instructors, by using breakpoints and monitoring the values of certain variables, can focus on different aspects of the processing and drive the sessions according to the concepts taught in the lectures.

While these methodologies are mainly focused on the implementation aspects of language processors, we have realized from our experience in teaching this topic at the Complutense University of Madrid that, in addition to emphasizing implementation, it is necessary to emphasize specification aspects, as the specification of language processors is a critical concern throughout the whole development process. Thus, strategies strongly geared toward implementation aspects are not capable of successfully helping students understand the formal specifications taught in lectures. Our main aim with *Evaluators* is to palliate this inconvenience from the beginning, teaching the basic concepts of attribute grammars, which is a specification formalism that fits well with the syntax-directed translation paradigm that we encourage at the Complutense University of Madrid.

2.2. Visualization and simulation tools for teaching Computer Science

The main objective of the serious games generated with *Evaluators* is to assist students of Compiler Construction in their learning by immersing them in the semantic evaluation process involved in attribute grammars. To this end, *Evaluators* games are similar to

other visualization and simulation tools used to support the teaching and learning of Computer Science, for example, the *Query Simulation System*, an educational simulator designed to enhance learning about the internal process that occur when querying a database (Allestein, Yost, Wagner, & Morrison, 2008); the *Java Execution Simulator* (JES), a Java simulator for a reduced subset of the language (Robbins, 2007); the *UNIX Concurrent I/O Simulator*, educational software that can run C programs and that shows different textual information to illustrate the concurrent reading and writing of files (Robbins, 2006); the *Address Translation Simulator*, a software tool for enhancing the learning of the process implied in the address translation in operating systems (Robbins, 2005); and, finally, the *Animation of linked lists in Java*, a library for substituting predefined linked lists with new lists whose behaviors can be viewed (Desherm, MacFall, & Uti, 2002).

All of these visualization and simulation tools that address different topics enrich the educational experience of students by offering visual illustrations and animating the processes and algorithms presented in lectures. Furthermore, some of these tools, such as the *Address Translation Simulation*, track the actions taken by the user so that the instructor can analyze them and target his/her lectures to address students' misconceptions and other problems encountered by students. Serious games generated with *Evaluators* largely include these two main characteristics and, in addition, encourage the active engagement of students by letting them interact with, and become part of, the simulation process.

2.3. Educational tools for the teaching and learning of Compiler Construction

Teaching and learning Compiler Construction can be supported by different educational tools focused on different aspects of the development process. These tools can be roughly classified in the following categories:

- *Theoretical machine simulators*. Tools in this category provide a dynamic and amusing way to present the theoretical machines underlying language processors, as well as their associated algorithms. Traditionally, these concepts are presented to students in a poor and confusing way using slides or the blackboard. However, the animation of the machines and the possibility for students to experiment with these simulations enhance the learning experience. The following are examples of such tools: jFAST (White & Way, 2006) is a simulator for (finite or stack) automata and theoretical machines (such as a Turing machine), and FSA Simulator (Grinder, 2002; Grinder, 2003) is a tool that is similar to jFAST but is capable of comparing two automata and extracting the differences between them, which is a useful feature for students so that they can compare their automata with those provided by the instructor.
- *Analysis algorithm viewers*. These tools can illustrate the performance of different syntactic-semantic analysis algorithms so that students can better understand these algorithms and the purpose of the data structures implied. The tools achieve this by using abstract representations of the data structures and letting the student visualize the algorithms' performance. Example of these tools include BURGRAM (Garcia-Osorio, Gomez-Palacios, & Garcia-Pedrajas, 2008), which emulates ascent and descent syntactic analysis algorithms; SEFALAS (Jodar-Reyes & Revelles-Moreno, 2010), which can simulate different lexical and syntactic analysis methods; and CUPV (Kaplan & Shoup, 2000), a visual debugger for the CUP parser generation tool that is able to show the analysis stack and its behavior during the parsing of a sentence.
- *Compilation process structure viewers*. These tools show students the construction, update and query of the different data structures implied in compiling, such as the parse tree or the symbol table. Some remarkable examples of this type of software include SOTA (Urquiza-Fuentes, Gallego-Carrillo, Gortazar-Bellas, & Velazquez-Iturbide, 2006), which is focused on the symbol table; the Tree-Viewer Library (Vegdahl, 2000), which produces syntactic tree visualizations when included in the students' compiler projects; the Annotating Debugger (Vegdahl, 2000), which lets students visualize the stack machine that would execute the object code generated by the language processor; and Proletool (Castro-Schez, Redondo, Gallardo, & Jurado, 2012), which is a tool that lets students describe, using formal languages, both problems and solutions concerning typical processes that are involved in the construction of parsers, as well as visualize the solutions and test whether the provided solutions are correct.
- *Generic development tools for language processors*. These tools can generate language processors from high-level specifications of these languages. These tools are widely used for the professional development of compilers, but they also can be used for educational purposes in Compiler Construction courses. Examples of these tools that have been used for educational purposes include LISA (Mernik & Zumer, 2003), a tool based on attribute grammars, and ANTLRWorks, a graphic-integrated environment for the ANTLR parser generation tool (Parr, 2007).
- *Generative environments of compiler viewers*. These environments can be considered educational specializations of the aforementioned generic development tools. These environments let students develop language processors from their specifications, but in this case, the generated processors are able to generate educationally geared visualizations of the analysis process. Some examples of this type of environment include VCOCO (Resler & Deaver, 1998), which extends the COCO parser generator tool to generate processors capable of displaying the performance of each processing stage, and PAG (Sierra et al., 2008), which generates language processor prototypes from attribute grammar specifications that are capable of analyzing sentences and generating parse-tree viewers.

In summary, most of the tools presented in this section are focused in the implementation phase of compiling, even those related to theoretical machines, because these machines are abstract models of the final components of compilers. Indeed, among all the mentioned tools, only PAG and LISA are based on high-level specification formalisms: attribute grammars. However, these tools are able to cover more advanced stages in the development process, which require a critical repertoire of previously acquired skills. Thus, a gap can be identified in the tools for acquiring these basics skills for syntax-directed processing based on high-level specifications. *Evaluators* is conceived as an instrument to fill this gap.

2.4. Educational games for Computer Science Education

Evaluators is an educational system based on serious games for learning basic concepts about language processors. The use of educational games is a growing trend in the field of e-learning (Amory et al.,1999; Gee, 2003; Minua et al., 2011), but it is not a new idea. These games (not only computer games) have been traditionally used to teach different concepts and skills. Computer games let users immerse themselves in a virtual world designed to teach a range of processes and techniques, in a realistic way (Shaffer, Squire, Richard, & Gee, 2005), and they include four key characteristics that make them unique learning tools: computer games are fun, they are immersive, they encourage cooperation and competitiveness, and they stimulate the creation of user communities.

Due to these features, computer games are a powerful tool for education.

Thus, education in Computer Science has also adopted computer games to enhance teaching and learning in different disciplines. Representative examples include the Age of Computers (Natvig & Line, 2004) and Nintendo DS I/O game (Larraza-Mendiluze & Garay-Vitoria, 2010) in the domain of computer architecture; Wu's Castle (Eagle & Barnes, 2008) in the domain of basic programming; ViRPlay (Jiménez-Díaz et al., 2007) in the domain of software design; SimSE (Navarro & Hoek, 2009) in the domain of software engineering and project management; and JV^{2}M (Gómez-Martín, Gómez-Martín, & González-Calero, 2006) in the domain of Compiler Construction. An interesting feature of all these serious games is the possibility of providing users with instant feedback when they perform an action in the virtual environment. Additionally, such games are very attractive to students, as claimed, for instance, by Larraza-Mendiluze and Garay-Vitoria (2010).

Evaluators is part of this initiative to use serious games in Computer Science education. It shares with the system JV^{2}M the learning domain of Compiler Construction. However, contrary to JV^{2}M, which is oriented to teaching students to compile Java programs into bytecode, *Evaluators* games are focused on more fundamental aspects of syntax-directed translation and its specification.

3. Evaluators

To describe the essence of the educational system *Evaluators*, we will start by introducing attribute grammar formalism (Section 3.1). Then, we will present the system workflow (Section 3.2). Finally, we will describe in detail each of the software subsystems that compose *Evaluators*: the authoring tool (Section 3.3), the customizer (Section 3.4), the generated serious games (Section 3.5) and the analytic tool (Section 3.6).

3.1. Attribute grammars

Attribute grammars were developed by Donald E. Knuth as a formalism for providing declarative descriptions of language processing tasks (Knuth, 1968). For this purpose, attribute grammars adhere to a syntax-directed language processing style, according to which the processing of the sentences of a language is driven by the syntactic structure of these sentences.

Knuth's proposal classified attribute grammars as an extension of *context-free grammars*. A context-free grammar describes the syntactic structure of a language by means of syntactic rules. Syntactic rules are composed of *terminal* symbols, which are the primitive constructs of the language described, and *non-terminal* symbols, which constitute the syntactic variables denoting the sets of sentences. A syntactic rule specifies how a non-terminal symbol can be structured as a sequence of other non-terminal and terminal symbols. For instance, in Fig. 1a (which shows an example of a context-free grammar for very simple arithmetic expressions) *num*, *iden* and + are examples of terminal symbols (these symbols represent, respectively, numbers, identifiers and the *addition* operator), *expr* and *term* are examples of non-terminal symbols (they represent, respectively, additive and basic arithmetic expressions), and *expr* → *expr* + *term* is a syntactic rule specifying that an expression may be constituted by another expression followed by the + operator and a *term*. Syntactic rules define the hierarchical structure of each sentence of the language, which is called a sentence *parse tree*. This tree-shaped structure shows how to apply the syntactic rules to generate a sentence. In parse trees, internal nodes are labeled with non-terminal symbols, leaf nodes are labeled with terminal symbols, and parent–child relationships are determined by the application of syntactic rules (in Fig. 1b, the reader can see the parse tree for the sentence $x + 5 + y$, where x and y are identifiers, and 5 is a number).

Context-free grammars do not go beyond assigning these tree-shaped structures to sentences. Building on context-free grammars, Knuth added processing description capabilities to the formalism by associating semantic attributes with the syntactic symbols and semantic equations with the syntactic rules (Fig. 1c shows an attribute grammar based on the context-free grammar depicted in Fig. 1a):

- Semantic attributes hold the processing results for the symbols with which they are associated. They are a type of placeholder in which to store the values obtained during the processing of a sentence. Semantic attributes can be classified into different groups according to the type of information that they hold. There are three types of attributes: synthesized, lexical and

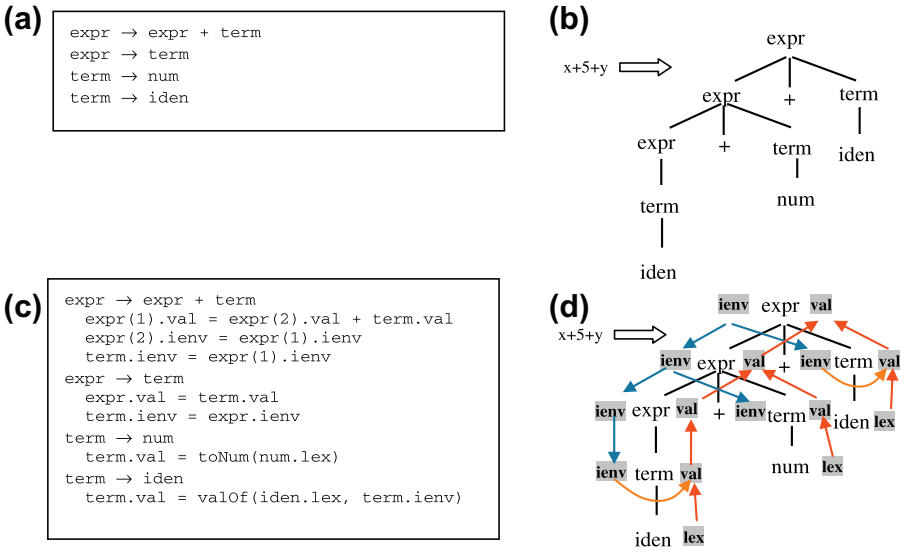


Fig. 1. (a) An example of a context-free grammar for simple arithmetic expressions, (b) the parse tree associated with the sentence $x + 5 + y$, (c) an example of an attribute grammar for simple arithmetic expressions that extends the context-free grammar in (a), (d) the attributed parse tree associated with the sentence $x + 5 + y$, along with the dependencies between attributes.

inherited. Synthesized attributes hold information that is related to the meaning of non-terminal symbols (e.g., the synthesized attribute *val* of the non-terminal symbols *expr* and *term* in Fig. 1c represents their respective values). Lexical attributes play the same role as synthesized ones but are associated with terminal symbols (e.g., in Fig. 1c, *lex* is a lexical attribute representing the actual number or identifier). Finally, inherited attributes contain the additional contextual information needed to compute the values of the synthesized attributes (e.g., in Fig. 1c, *ienv* represents an environment assigning values to identifiers).

- Semantic equations, in turn, indicate how to use the semantic attributes in the syntactic rules to compute other semantic attributes (e.g., the first semantic equation of $expr \rightarrow expr + term$ in Fig. 1c, $expr(1).val = expr(2).val + term.val$, indicates how to compute the value of the additive expression by adding the values of its operands).

With this extension, Knuth endowed attributed parse trees (i.e., parse trees with the semantic attributes for each node) with semantic evaluation (see in Fig. 1d the attributed parse tree for the sentence $x + 5 + y$). Semantic evaluation consists of the calculation of every attribute of the attributed parse tree. This process is driven by semantic equations, which establish the dependency relations among attributes and determine which attribute values are needed to calculate which other attribute values (in Fig. 1d, arrows represent these dependency relations; notice that if an arrow goes from attribute $a1$ to $a2$, it means that $a1$ is used to compute $a2$). Therefore, the calculation of the attributes' values must satisfy the implicit constraints defined by the semantic equations. Indeed, before computing the value of an attribute, the values of the attributes on which it depends should already have been computed. When this constraint is satisfied, the evaluation order is irrelevant.

The characteristics of attribute grammars turn them into a formalism that is very suitable for the specification tasks in computer language implementation (Paakki, 1995); this formalism is even better than other sophisticated formalisms such as translation schemes – context-free grammars with interleaved semantic actions that fire during parsing (Aho et al., 2007). Based on our routine experience teaching computer language implementation, we have confirmed that attribute grammars promote the modularity, extensibility and maintainability of the language processor specifications and their subsequent implementations. Thus, they have become an essential tool in computer language implementation courses. Unfortunately, we have also detected that students encounter difficulties in understanding this type of formalism, especially those not accustomed to dealing with formal and declarative specification methods. Thus, our main aim with *Evaluators* was to help students overcome these difficulties.

3.2. An overview of Evaluators

Evaluators is an educational system that involves both instructors and students. The system provides instructors with different tools: an authoring tool for creating exercises concerning processing tasks formalized as attribute grammars, a customizer tool for generating the serious games from the exercises, and an analytic tool to evaluate student performance on these games. In addition, the serious games developed engage students in motivating ways to solve the exercises created by the instructors. Fig. 2 outlines the ways that both types of actors use the different *Evaluators* subsystems (Rodriguez-Cerezo, Gómez-Albarrán, and Sierra (2011) describe the process model that was followed in the construction of *Evaluators*). According to this schema,

- The instructor first elaborates sets of suitable exercises for the students by using the *Evaluators authoring tool*. These exercises conform to the structure of the exercises described in Section 1 and include the following components: an informal description of the language processing task, a formal description of this task (the attribute grammar), a sentence to be processed and the attributed parse tree of the sentence. As we will see in the next section, the current version of the authoring tool reduces the exercise creation workload to a great extent by automatically deriving some parts of the exercise from the formal description.
- Then, when the exercises are ready, the instructor can use the *Evaluators customizer tool* to generate the game to be used by the students. For this purpose, he/she first creates an

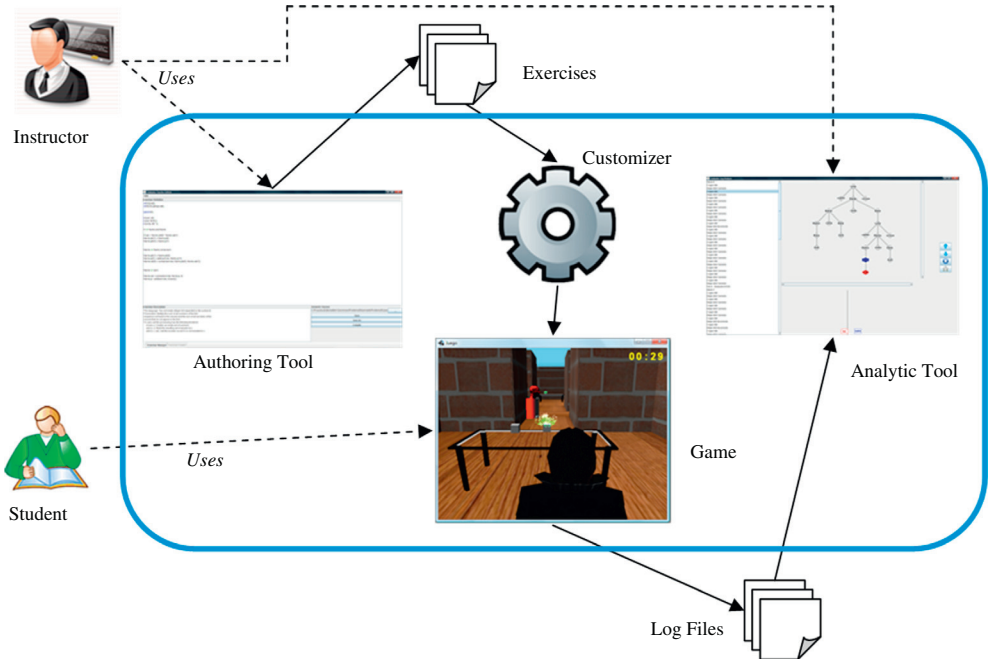


Fig. 2. *Evaluators* workflow.

assignment by selecting a suitable set of exercises. Then, the customizer tool automatically processes the information for each exercise and generates a game in which each exercise corresponds to a level of the game.
- Once the game is created, it is deployed to the students, who play it. The students have to complete each level of the game (i.e., they have to answer each corresponding exercise) to complete the assignment. The game gives them appropriate feedback while they play each level, and it also logs their activity.
- To complete the process, the instructors use the *Evaluators analytic tool* to inspect the log files. The analytic tool provides a visual environment in which the instructors can visualize the attributed parse tree and explore the actions performed by the students, checking their correctness and the time elapsed in solving each level. As a result, they can detect and analyze the mistakes that the students make while completing the exercises. This way, the instructors can subsequently plan personalized reinforcement actions to help their students better understand the least-comprehended concepts.

The next subsections explain, in detail, the different tools of the *Evaluators* system and the serious games deployed by *Evaluators*.

3.3. The authoring tool

To facilitate the first step of creating serious games, *Evaluators* provides an authoring tool that the instructor uses to define processing task problems (exercises) whose structures conform to the one described in Section 1. The tool has two main features:

- The *attribute grammar manager*. This component enables the creation, edition and deletion of the attribute grammars that formally define the language processing tasks. Thus, the instructors can easily maintain a library of attribute grammars for his/her processing task problems, reducing the cost of time required to create the exercises. To create and edit the grammars, the attribute grammar manager provides an appropriate editor to write both the informal description of the processing task and the formal description in terms of the actual attribute grammar (Fig. 3a shows a snapshot of this editor). To describe attribute grammars, we developed a domain-specific language similar to that usually used in the lectures and books that are found in Compiler Construction courses.
- The *exercise creator*. This component makes it possible to create collections of exercises. For each exercise in the collection, the instructor must select the attribute grammar describing the language processing task that he/she has previously defined using the attribute grammar manager and must specify the sentence to be processed. Contrary to previous versions of *Evaluators* (Rodriguez-Cerezo et al., 2011, 2012), in the current, enhanced version of the exercise creator, the instructor does not need to provide the parse tree of the sentence, which is a rather tedious and error-prone task. The parse tree is then automatically derived from the formal specification (the attribute grammar) provided by parsing the sentence according to the grammar, and the nodes of the tree are decorated with semantic attributes. To test whether an action made by a student while playing the game is correct, it is necessary to know the dependencies among the semantic attributes. To provide better assistance to students while they play the game, it is necessary to know the values of the attributes. Again, the current version of the exercise editor automatically derives both attribute dependencies and attribute values from the attribute grammar and the sentence, which substantially reduces the instructor's effort that is required to develop the exercise collection. In addition, to allow instructors to focus the processing problem of the exercises on particular processing aspects (e.g., translation of control sentences), the exercise creator also makes it possible to set some attributes to *solved* or *already computed* so that, in the game, students do not have to determine how to compute them. Finally, the exercise creator makes it possible to set other features of the exercise, such as the maximum number of mistakes that the student is allowed to make while solving it. Fig. 3b shows a snapshot of the exercise creator.

3.4. The customizer

Once the instructor creates a collection of exercises using the authoring tool, he/she must transform the collection into a serious game. This is performed using the customizer tool. For this purpose, each exercise in the collection is transformed into a level of the game using a metaphor that adopts the following conventions:

- The student is immersed in a maze-like world whose structure is based on the parse tree of the exercise. With this idea, the tree nodes are transformed into rooms in the maze (Fig. 4a). These

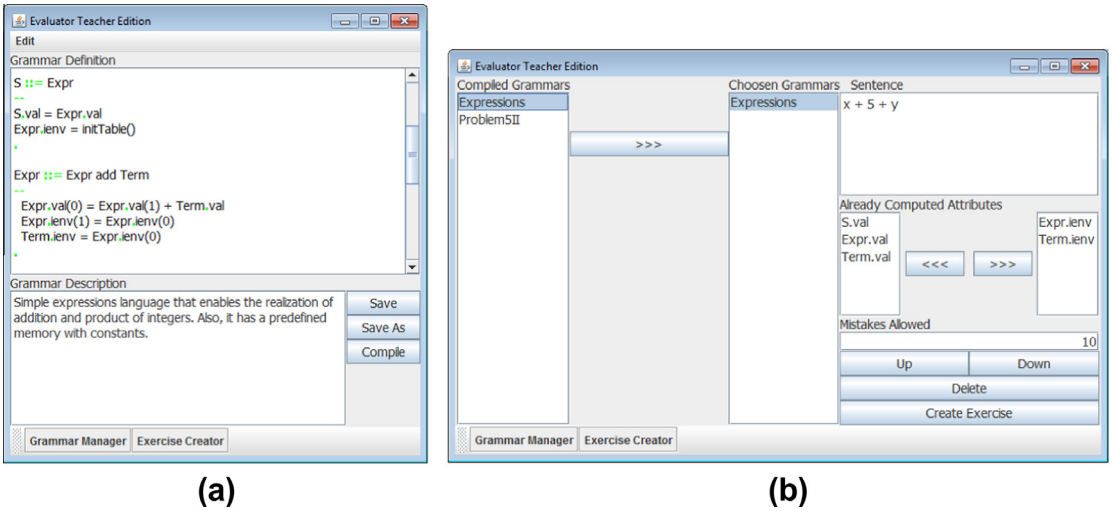


Fig. 3. (a) The attribute grammar manager of the authoring tool and (b) the exercise creator of the authoring tool.

rooms are connected by halls with incoming and outgoing corridors, yielding a visual tree-like representation of the syntactic rule instances (Fig. 4b).
- Inside each room, the student can find a table with boxes on it (Fig. 4c). These boxes represent the instances of the semantic attributes associated to the node corresponding to the room. The boxes can contain objects that correspond to the values of the attributes (Fig. 4d).
- Finally, the student is represented in the game with an avatar that can move through the maze and carry objects. The student has to move the correct objects from box to box, reproducing the attribute evaluation process (Fig. 4e).

3.5. Evaluators games

Once the game is generated, instructors deploy it to the students. Students complete each level (i.e., solve the corresponding exercise) of the game, which entails following a semantic evaluation strategy that is appropriate for the language processing problem in the level. To complete the level, the student must command the avatar through the maze, filling each box (i.e., each attribute instance) with the corresponding objects from other boxes (i.e., values from other attribute instances). This process is equivalent to deciding, for each non-computed attribute, which attributes are needed to compute its value. To facilitate this process, the game provides the following features:

- A box can adopt different visual appearances to indicate the state of computation of the corresponding attribute. A box that does not emit any light corresponds to an attribute that has not yet been computed (Fig. 5a). A box that emits a beam of multi-colored light corresponds to an attribute whose value has been calculated (Fig. 5b).
- The game notes the student's mistakes by profiting from the information about the dependencies among attributes that the exercise creator tool automatically extracts for each exercise. When a student makes a mistake (i.e., he/she drops an object in the wrong box), the box emits a plume of grey smoke (Fig. 5c). The mistake is also indicated with a textual message. If the object was dropped into the correct box, the visual effect depends on whether the evaluation process for the attribute has been completed or not. If the evaluation process has been completed, the box will emit the multi-colored beam previously described. In the other case, the box will remain inactive. In both cases, the system will show a textual message about the evaluation process of the attribute.
- The game keeps track of the number of mistakes made by the player, forcing him/her to restart the level if he/she exceeds a certain number of mistakes set by the instructor in the corresponding exercise description.

Furthermore, the game provides the following additional features to support the students while playing:

- Emerging panels with information about the attributes, such as the name, number of still-unsolved dependencies, and, when solved, their values, which are automatically derived by the authoring tool.
- A screen with a description of the exercise (Fig. 5d), which the student can consult at any point during the game and contains the informal definition and the attribute grammar established by the instructor during the creation of the exercise.
- A global view of the maze (Fig. 5e).
- An oracle in each hall (Fig. 5f) of the maze that the player can consult to obtain information about the rule associated with the hall.

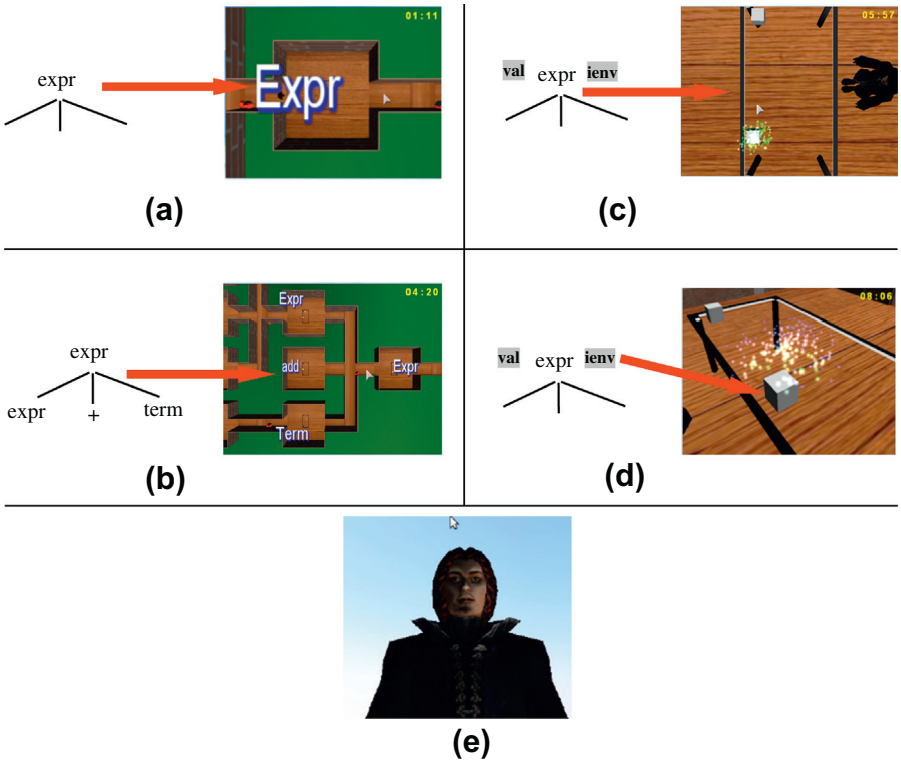


Fig. 4. Correspondence between an attributed parse tree and the elements in the games generated with *Evaluators*.

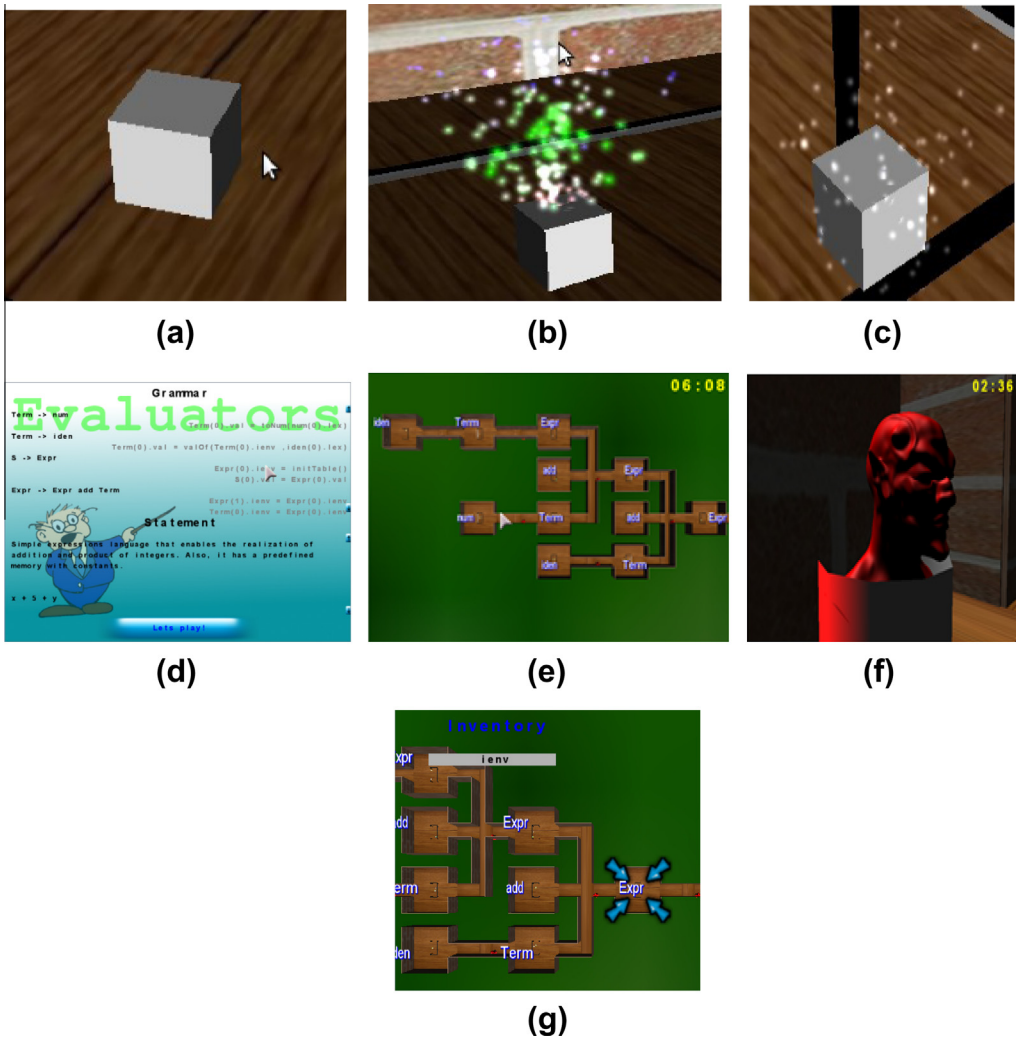


Fig. 5. Screenshots of a game deployed by *Evaluators* (see main text for further explanations).

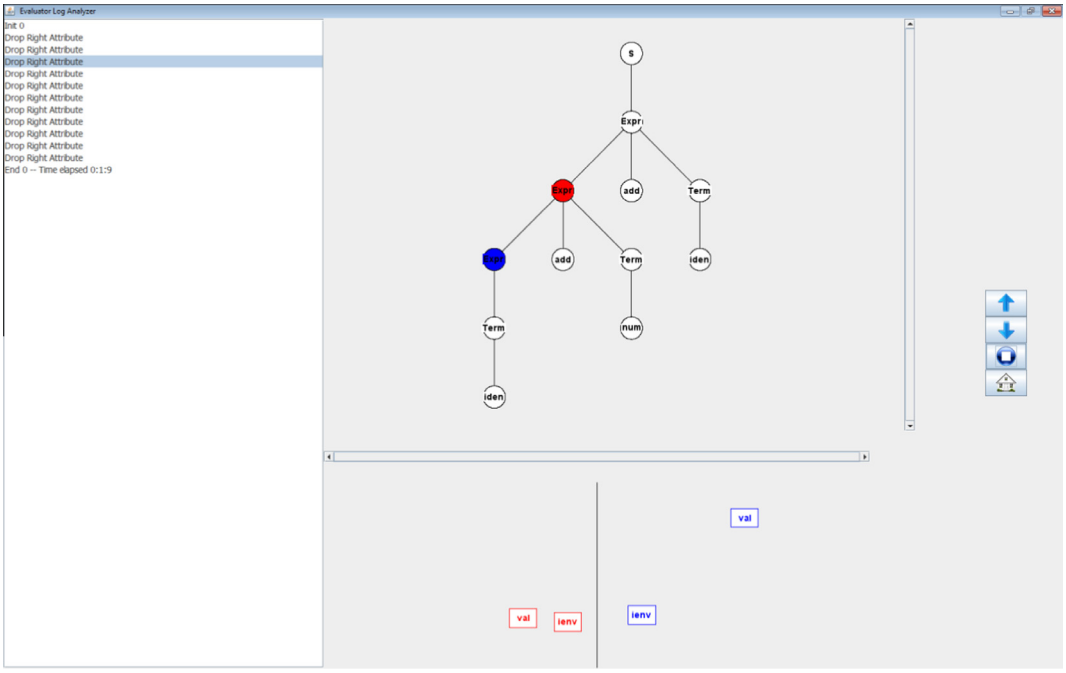


Fig. 6. The *Evaluators*' analytic tool.

To better control the evaluation process, the avatar has an associated inventory in which it can store the objects (i.e., the values of the attributes) obtained from the activated boxes. To facilitate the identification of the objects in the inventory, the game can display a global view of the maze (parse tree), highlighting the room (node) that contains the box (attribute) from which the player obtained the object (attribute value) (Fig. 5g).

Finally, as stated before, the game generated with *Evaluators* can register, in log files, the actions performed by the students and also whether they restarted any levels. As we will see in the next section, these log files can be visualized in an attractive way using the analytic tool of *Evaluators*.

3.6. The analytic tool

The analytic tool in *Evaluators* lets instructors see a basic replay of the actions performed by the students in each level. This way, the instructors can assess the gameplay of the students, evaluate their performance on *Evaluators*' games and detect potential misconceptions that could be solved during later interactions with the students.

Student actions on each level are shown in a global view of the corresponding parse tree. Instructors can explore the actions performed by the students on such a parse tree and the attribute values that the students moved (see Fig. 6). The tool also shows the time elapsed in the resolution of each level.

The most remarkable aspect of the analytic tool is its visual impact and the easiness with which instructors, with a glance, can visualize the actions performed by the student that are relevant to the evaluation process (these actions are translated from game terms into the parse trees associated with the levels of the game).

4. System assessment

In this section, we assess the use of *Evaluators* in a 90-h Compiler Construction course at the Complutense University of Madrid, which has a mean of 75 students enrolled per year. We start by reviewing the results collected from an opinion survey on *Evaluators* conducted with instructors in the 2010–2011 academic year (Section 4.1). Then, we present the results obtained from an opinion survey conducted among students during the same academic year (Section 4.2). Finally, in Section 4.3 we analyze the educational effectiveness of *Evaluators* in terms of the grades obtained by the students in the last five academic years (the three first years did not include the use of *Evaluators*, whereas the last 2 years did).

4.1. Instructor opinions

When we finished developing *Evaluators*, we conducted an opinion survey with Computer Science instructors at the Complutense University of Madrid to assess the system tools. Ten instructors were involved in the experience, whose areas of expertise ranged from language processing and introductory programming to educational software development. Before the interview, we explained to them the fundamentals of the learning domain (for those instructors not familiar with attribute grammars), as well as the basic ideas in *Evaluators*. A demo of the different parts of the system was presented, and then the instructors could use the different elements in *Evaluators*.

We elaborated a set of open questions for the survey concerning the following different aspects of the system (Table 1 shows the questions):

- The expected utility for the students and the adequacy for Compiler Construction courses (questions 1, 2 and 3).
- Authoring tool usage and the appropriateness of the exercise definitions (questions 4 and 5).
- Analytic tool usage (question 6).

The following main findings were obtained with this informal survey:

- All the instructors agreed that the serious games generated in *Evaluators* have a potential educational utility due to their heavy motivational component. In addition, the instructors found the exercise-driven approach to be a natural way to generate the games. Although the idea of transforming the parse tree into a maze world seemed to be very appropriate to the instructors, they found some deficiencies in the generated games, including a lack of feedback presented that could hinder students in figuring out what the state of the solution was during game play. Additionally, some of the instructors were interested in using similar games for other Computer Science course topics (e.g., introductory programming courses).
- Concerning the authoring tool, the exercise-driven approach to the construction of educational serious games was strongly appreciated by all the instructors interviewed. In particular, the tool was perceived as very useful by the instructors familiar with the Compiler Construction topic. Indeed, some of the instructors interviewed who had collaborated in the assessment of the previous version of *Evaluators* (Rodriguez-Cerezo et al., 2011, 2012) were very pleased with the improvements made to the authoring tool. They especially appreciated the ease with which the exercises could be created quickly and turned into serious games in comparison to the previous version of tool (as stated before, the previous authoring tool did not generate the attributed parse tree automatically from the problem description, so the instructor had to specify it, decorate it with the attribute instances and specify the attribute dependency graph, which resulted in a prohibitively heavy workload for instructors engaged in the creation of the serious games).
- Finally, concerning the analytic tool, they strongly appreciated the presentation of the actions performed by the students in the parse trees associated with each level of the game, although they also highlighted the possibility of augmenting the types of actions registered by the game to inform the instructor of student inactivity or erratic behavior.

In summary, the instructors had a positive perception of the system. They found *Evaluators*' serious games to be useful instruments in facilitating the teaching and learning of basic concepts of attribute grammars. In addition, they perceived the extrapolation of the ideas to other areas of Computer Science education to be feasible and recommendable. They perceived the analytic tool to be a useful artifact for monitoring the progress of their students. Finally, and more importantly, they perceived the approach to the

Table 1
Some questions asked to the instructors in the interviews.

Question 1	Do you think *Evaluators* can help students understand how attribute grammars work?
Question 2	Would you use *Evaluators* for teaching the essentials of attribute grammars in a Compiler Construction course?
Question 3	Which are the most outstanding aspects of the games generated with *Evaluators*? Which are the least outstanding aspects?
Question 4	Do you think that the authoring tool is easy and intuitive to use? How would you improve the authoring tool?
Question 5	Does the idea of defining the exercises by using grammars and language sentences seem appropriate to you?
Question 6	Do you think that the analytic tool allows teachers to easily interpret the solutions proposed by the students? How would you improve the analytic tool?

generation of serious games adopted in *Evaluators* to be successful and cost-effective. In addition to this positive assessment, the instructors also found several aspects that could be improved in a new version of *Evaluators* (e.g., better feedback in generated games, deeper support for student behaviors in the analytic tool).

4.2. Student opinions

To assess the perceived utility among students of the serious games generated by *Evaluators*, we collected opinions offered voluntarily by 19 out of the 75 students who used the system during the 2010–2011 academic year. After these students played and completed all the levels of the game generated, we collected their opinions with a survey consisting of seventeen questions that are answered using a four-point Likert scale (Disagree – Slightly Disagree – Slightly Agree – Agree). The questions were adapted from the TUP model, which is designed for evaluating educational software (Bednarik, Gerdt, Miraftabi, & Tukiainen, 2004), and were divided into groups addressing different aspects of the software, such as the pedagogical utility of the games or interactions. The left column of Table 2 lists the questions; the right column shows the percentage of students who selected "Slightly Agree" or "Agree" for each question on the survey.

The results obtained were mostly satisfactory. Among the students surveyed, the serious games produced with *Evaluators* were considered useful as complementary instruments to the lectures. A high percentage of the students found the games generated to be enjoyable and useful for learning the different aspects of attribute grammars (i.e., semantic attributes, semantic equations, and the processing model). Notice also that the students perceived the games to be helpful for understanding the differences between the three types of attributes in attribute grammars. Most of the students thought, as the instructors did, that representing the parse tree with a maze was a straightforward metaphor. However, there was a divergence of opinions concerning the use of tables to collect attribute information. We consider it very positive that students had an accurate perception of the state of problem solving while playing the game. All of this feedback, together with the considerable percentage of students who would recommend the use of *Evaluators* to their classmates, makes us optimistic about the success of *Evaluators*.

We were also aware of some aspects of the games that need to be improved in a later version of *Evaluators*. The games do not have a very steep learning curve, but a considerable number of students had usage difficulties during the experience. Some of these difficulties might be solved by distributing detailed demos of the game play along with the game. We are also conscious that the feedback the students receive when they make incorrect actions should be improved. The description of the problem was frequently consulted by the students, but the oracles were rarely used. The oracles should be improved, and we should even reconsider whether they are necessary. Finally, the survey shows that the information associated with the objects in the inventory is sometimes insufficient. We plan to test whether introducing additional information in the textual panels associated to the attributes resolves this problem.

Table 2
Student agreement with survey items.

Survey item	% Agree
Pedagogical utility	
It helps me understand the processing model of attribute grammars	73.7
It helps me understand the role of semantic attributes	73.7
It helps me understand the role of semantic equations	68.4
It helps me understand the differences among the different types of grammatical attributes	73.7
I do not find it difficult to understand my own mistakes when using the tool	36.8
Context adequacy	
Evaluators suitably complements the lectures	84.2
Evaluators motivates me to study attribute grammar processing	52.6
Learnability	
I can rapidly start working with the tool without a long period of training	63.2
Mazes are a straightforward metaphor for representing attributed syntax trees	73.7
Using tables to collect information about attributes seems natural to me	52.6
Interaction	
I have used the problem statement several times	68.4
I have used the oracles	10.5
It is easy to identify which attribute instance corresponds to each attribute copy in the inventory	31.6
It is always easy to identify the state of problem solving	73.7
Overall usability	
Evaluators is easy to use	42.1
I enjoyed using *Evaluators*	73.7
I would recommend *Evaluators* to other Compiler Construction students	68.4

4.3. Educational effectiveness study

During the last two academic years (2010–2011 and 2011–2012), *Evaluators* has been used to support the aforementioned Compiler Construction course. We have collected the academic results of the students enrolled in this course in these 2 years to compare them with the results of the students in the three previous years, when *Evaluators* was not used. Fig. 7 depicts the mean grades of the years 2007–2008, 2008–2009 and 2009–2010, when Evaluators was not used, and the last 2 years, 2010–2011 and 2011–2012, when *Evaluators* was used. More precisely,

- In the first 2 years, 2007–2008 and 2008–2009, the instructors followed a traditional pedagogical method, based on lectures and a final project about Compiler Construction.
- In the year 2009–2010, lectures were reinforced with batteries of exercises related to attribute grammars (as described in Section 1) that were solved by the students using paper and pencil. Students also had to develop a final project on Compiler Construction.
- Finally, in the two last years, 2010–2011 and 2011–2012, the instructors advocated for a strategy based on reinforcing the basic concepts of syntax-directed translation using *Evaluators*. A final project on Compiler Construction was assigned.

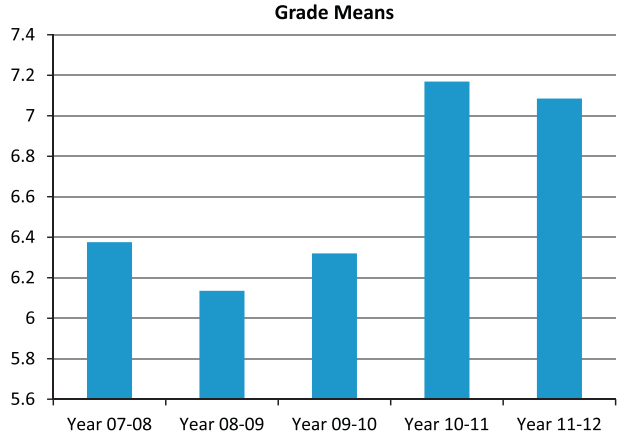


Fig. 7. Final mean grades in the Compiler Construction course from the academic years 2007–2008 to 2011–2012.

Table 3
Statistical significance coefficients obtained from the Mann–Whitney statistical test. Italicized values correspond to *p*-values supporting significant differences in the grades resulting from different editions of the Compiler Construction course.

	07–08			
08–09	0.295	**08–09**		
09–10	0.853	0.304	**09–10**	
10–11	*0.017*	*0.000*	*0.007*	**10–11**
11–12	*0.021*	*0.000*	*0.005*	0.726

As shown in Fig. 7, the mean grades in the years prior to 2010–2011 are approximately six points out of 10. The mean grades in the following years increased by approximately one point with respect to the previous years. To test these differences, we ran a Kruskal–Wallis test on the five sets of grades and found significant differences (*p*-value = *0.000*). A post hoc Mann–Whitney analysis supported these differences. The significance of the differences between each pair of years is summarized in Table 3. In this table, the italicized cells correspond to *p*-values that evidence a violation of the null hypothesis according to the Mann–Whitney test and therefore indicate significant differences in grades. The non-italicized cells correspond to pairs of years whose grades are non-significantly different. Thus:

- The results show that there are significant differences neither among the grade distributions of the first three academic years (2007–2008, 2008–2009 and 2009–2010, when *Evaluators* was not used) nor among the grade distributions of the two last years (2010–2011 and 2011–2012, when *Evaluators* was used).
- However, the differences among the grade distributions of the last 2 years and the first 3 years are statistically significant.

Thus, the results obtained were highly satisfactory and motivating. Indeed, once *Evaluators* and its associated pedagogical strategy were adopted in our Compiler Construction course, we observed a significant enhancement in our students' performance. In particular, in the 2010–2011 academic year the adoption of *Evaluators* resulted in a remarkable increase of the mean grades (about one point with respect to the previous years). This enhancement was revalidated in the 2011–2012 academic year, in which we obtained similar results. In addition, the perceived differences between the use of *Evaluators* (2010–2012 period) and the adoption of more traditional orientations (2007–2010 period) were proven statistically significant. In our opinion, the reasons for these benefits are twofold:

- On the one hand, the pedagogical strategy behinds *Evaluators* helped our students to understand the basic principles of attribute grammars. The need to explain, on concrete sentences, how to carry out the propagation of attribute values forced students to *read* and *understand* representative examples of attribute-grammars before using the formalism to produce their own specifications, and also to grasp the fundamental aspects of the formalism (like the dependency-driven computation style).
- On the other hand, *Evaluators* promoted the active participation of students in an interactive simulation of the semantic evaluation process. In particular, students got feedback on the actions they carried out (they were warned when they made erroneous movements and rewarded when they made the correct ones). This aspect was, in our opinion, decisive to make the potential of the pedagogical strategy effective. Indeed, although during the 2009–2010 academic year students solved exercises oriented to understand basic syntax-direct translation concepts that were similar to the exercises used in *Evaluators*, the benefits were not made apparent in the results: the final mean grade in 2009–2010 was very similar to those in 2007–2008 and 2008–2009 academic years, when a more traditional approach was adopted.

These results provide evidence in favor of the educational effectiveness of *Evaluators* and, consequently, of the strategy this system supports.

5. Conclusions and future work

This paper has presented *Evaluators*, an exercise-driven learning system that is focused on the basic concepts of attribute grammars and their underlying evaluation process. The system provides the necessary tools to let instructors create multilevel serious games from language processing exercises. These games are played by students to reinforce their comprehension of the basic concepts of attribute grammars and their computational model. The system also provides an analytic tool that lets instructors assess the performance of their students during the game. Thus, instructors can plan reinforcement activities to address the problems and deficiencies observed.

Additionally, we have presented different studies that assess the quality and utility of the system:

- Firstly, we have presented a qualitative study performed with Computer Science instructors at the Complutense University of Madrid. As a consequence we realize how instructors had a positive attitude towards the system, perceiving serious games generated by *Evaluators* as valuable instruments for facilitating the assimilation of attribute grammar fundamentals, and envisioning the generalization of the approach to other fields of Computer Science education. They also highlighted the usefulness of the analytic tool in order to assess student performance. But certainly the main result obtained during our evaluation with instructors was the common agreement on the exercise-driven generative approach implemented in *Evaluators* as a cost-effective approach for the development of serious games in Tertiary education. Indeed, instructors were delighted with the possibility of turning, with a single *click*, a collection of exercises in a working serious game without worrying about what is actually happening on the backstage.
- Then, we have presented a qualitative study conducted with Compiler Construction students, which evidenced the perceived utility of the serious games generated with *Evaluators* for learning basic concepts concerning attribute grammars. Students rated the games as useful and enjoyable instruments to practice with basic attribute grammar concepts, and they rated the metaphors adopted in these serious games as very natural ones. They also suggested some improvements (e.g., providing better feedback concerning wrong actions and more proactive oracles) that constitute the basis for our immediate future work.
- Finally, we have analyzed the educational effectiveness of the pedagogical strategy encouraged by *Evaluators*. To perform the analysis, we compared the grades from previous editions of an introductory course on Compiler Construction with the grades of the last two editions of the course, in which *Evaluators* and its underlying pedagogical strategy were incorporated to support the acquisition of basic skills for syntax-directed translation specified with attribute grammars. The analysis evidenced a significant improvement in the final grades. The use of the immersive strategy promoted by *Evaluators* serious games, which invites students to actively participate in the semantic evaluation process, was a key aspect to explain the improvement, as we do not observed similar enhancements by compelling our students to solve similar exercises by using paper and pencil.

For the immediate future, we are planning to address the deficiencies of the system detected during its assessment with the instructors and students. In particular, we are considering the inclusion of additional feedback for students when they perform incorrect actions that are based on the types of mistakes detected. We will improve *Evaluators'* usage documentation to reduce some difficulties found by the students. In this case, demos of different game plays could be helpful. Additionally, we are considering developing alternative metaphors to map the problems to serious games to reduce the amount of superfluous time that students spend while they play the games. Finally, we plan to carry out additional experiments involving both students and instructors to further improve the *Evaluators* tools and the generated serious games.

Acknowledgments

This research was partially supported by Project Grants TIN2010-21288-C02-01 and TIN2009-13692-C03-03. Daniel Rodríguez-Cerezo was supported by a Grant of the Spanish University Teacher Training Program (EDU/3445/2011). We also would like to thanks Rafael Fernández-López and Ángel Valero-Picazo by their invaluable contribution to the implementation of a previous version of *Evaluators*.

References

ACM/IEEE (2008). Computer Science Curriculum 2008: An Interim Revision of CS 2001.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, techniques and tools* (2nd ed.). Addison-Wesley.

Aiken, A. (1996). Cool: A portable project for teaching Compiler Construction. *ACM SIGPLAN Notice, 31*(7), 19–24.

Allestein, B., Yost, A., Wagner, P., & Morrison, J. (2008). A query simulation system to illustrate database query execution. In *Proceedings of the 39th SIGSE technical symposium on Computer Science education (SIGCSE'08)* (pp. 493–497). New York, NY, USA: ACM.

Amory, A., Naicker, K., Vincent, J., & Adams, C. (1999). The use of computer games as an educational tool: Identification of appropriate game types and game elements. *British Journal of Educational Technology, 30*(4), 311–321.

Baldwin, D. (2003). A compiler for teaching about compilers. In *Proceedings of the 34th SIGCSE technical symposium on Computer Science education (SIGCSE'03)* (pp. 220–223). New York, NY, USA: ACM.

Bednarik, R., Gerdt, P., Miraftabi, R. & Tukiainen, M. (2004). Development of the TUP Model – Evaluating Educational Software. In *Proceedings of the 4th IEEE international conference on advanced learning technologies (ICALT'04)* (pp. 699–701).

Bloom, B., Engelhart, M., Furst, E., Hill, W., & Krathwohl, D. R. (1956). *Taxonomy of educational objectives: The classification of educational goals. Handbook I, The cognitive domain*. David McKay Co Inc..

Castro-Schez, J. J., Redondo, M. A., Gallardo, J., & Jurado, F. (2012). Designing and developing software for educative virtual laboratories with language processing techniques: Lessons learned in practical experiments. *Journal of Research and Practice of Information Technology, 44*(3), 289–308.

de Freitas, S., & Liarokapis, F. (2011). Serious games: A new paradigm for education. *Serious Games and Edutainment Applications*, 9–23.

Desherm, H. L., MacFall, R. L., & Uti, N. (2002). Animation of Java linked lists. In *Proceedings of the 33rd SIGCSE technical symposium on Computer Science education* (pp. 53–57). New York, NY, USA: ACM.

Eagle, M., & Barnes, T. (2008). Wu's castle: Teaching arrays and loops in a game. In *Proceedings of the 13th annual conference on innovation and technology in Computer Science education* (pp. 245–249). New York, NY, USA: ACM.

Garcia-Osorio, S., Gomez-Palacios, C., & Garcia-Pedrajas, N. (2008). A tool for teaching LL and LR parsing algorithms. In *Proceedings of the 13th annual conference on innovation and technology in Computer Science education (ITiCSE'08)* (pp. 317). New York, NY, US: ACM.

Gee, J. P. (2003). *What video games have to teach us about learning and literacy*. New York, Basingstoke: Palgrave Macmillan (pp. 225). New York, Basingstoke: Palgrave Macmillan.

Gómez-Martín, M. -A., Gómez-Martín, P., & González-Calero, P. -A. (2006). Dynamic binding is the name of the game. In *Proceedings of the international conference on, entertainment computing (ICEC'06)* (pp. 229–232).

Grinder, M. T. (2002). Animating automata: A cross-platform program for teaching finite automata. In *Proceedings of the 33rd SIGCSE technical symposium on Computer Science education* (pp. 63–67). New York, NY, USA: ACM.

Grinder, M. T. (2003). A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. In *Proceedings of the 34th SIGCSE technical symposium on Computer Science education* (pp. 157–161). New York, NY, USA: ACM.

Jiménez-Díaz, G., Gómez-Albarrán, M., & González-Calero, P. -A. (2007). Pass the ball: Game-based Learning of Software Design. In *Proceedings of the international conference on, entertainment computing (ICEC'07)* (pp. 49–54).

Jiménez-Díaz, G., González-Calero, P.-A., & Gómez-Albarrán, M. (2012). Role play virtual worlds for teaching object oriented design: The ViRPlay development experience. *Software: Practice and Experience, 42*(2), 235–253.

Jodar-Reyes, J. -F. & Revelles-Moreno, J. (2010). SEFALAS: Software para la Enseñanza de las Fases de Análisis Léxico y Análisis Sintáctico. MSc. Project. University of Granada.

Kaplan, A., & Shoup, D. (2000). A visualization tool for generated parsers. In *Proceedings of the 31th SIGCSE technical symposium on Computer Science education (SIGCSE'00)* (pp. 11–15). New York, NY, USA: ACM.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical System Theory, 2*(2), 127–145.

Larraza-Mendiluze, E., & Garay-Vitoria, N. (2010). Changing the learning process of the input/output topics using a game in a portable console. In *Proceedings of the fifteenth annual conference on innovation and technology in Computer Science education (ITiCSE'10)* (pp. 316). New York, NY, USA: ACM.

Ledgard, H. F. (1971). Ten mini-languages: A study of topical issues in programming languages. *ACM Computing Surveys, 3*(3), 115–146.

Mernik, M., & Zumer, V. (2003). An educational tool for teaching Compiler Construction. *IEEE Transactions on Education, 46*, 61–68.

Minua, M., Andreas, O., & Lakhmi, J. (2011). *Serious games and edutainment applications*. Springer Verlag.

Natvig, L., & Line, S. (2004). Age of computers: Game-based teaching of computer fundamentals. In *Proceedings of the 9th annual SIGCSE conference on innovation and technology in Computer Science education (ITiCSE'04)* (pp. 107–111). New York, NY, USA: ACM.

Navarro, E., & Hoek, A. (2009). Multi-site evaluation of SimSE. In *Proceedings of the 40th ACM technical symposium on Computer Science education (SIGCSE'09)* (pp. 326–330). New York, NY, USA: ACM.

Paakki, J. (1995). Attribute grammar paradigms – A high-level methodology in language implementation. *ACM Computer Surveys, 27*(2), 196–255.

Parr, T. (2007). *The definitive ANTLR reference: Building domain-specific languages*. Raleigh (North Carolina): Pragmatic Bookshelf.

Resler, D., & Deaver, D. (1998). VCOCO: A visualization tool for teaching compilers. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on integrating technology into Computer Science education (ITiCSE'98)* (pp. 199–202). New York, NY, USA: ACM.

Robbins, S. (2005). An address translation simulator. In *Proceedings of 36th SIGCSE technical symposium on Computer Science education (SIGCSE'05)* (pp. 515–519). New York, NY, USA: ACM.

Robbins, S. (2006). A UNIX concurrent I/O simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer Science education (SIGCSE'06)* (pp. 303–307). New York, NY, USA: ACM.

Robbins, S. (2007). A Java execution simulator. In *Proceedings of the 38th SIGCSE technical symposium on Computer Science education (SIGCSE'07)* (pp. 536–540). New York, NY, USA: ACM.

Rodriguez-Cerezo, D., Gómez-Albarrán, M., & Sierra, J. L. (2011). From collection of exercises to educational games: A process model and a case study. In *Proceedings of the 11th IEEE international conference on advanced learning technologies (ICALT'11)* (pp. 282–284).

Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M. & Sierra, J. L. (2012). Facilitating comprehension of basic concepts in computer language implementation courses: A game based approach. In *Proceedings of the 14th international symposium on, computers in education (SIIE'12)* (pp. 29–31).

Ruckert, M. (2007). Teaching Compiler Construction and language design: Making the case for unusual compiler projects with postscript as the target language. In *Proceedings of the 29th SIGCSE technical symposium on Computer Science education* (pp. 232–236). New York, NY, USA: ACM.

Shaffer, D. W., Squire, K. R., Richard, H. & Gee, J. P. (2005). Video games and future of learning. *University of Wisconsin-Madison and Academic Advanced Distributed Learning Co-Laboratory*.

Shapiro, H. D., & Mickunas, M. D. (1976). A new approach to teaching a first course in Compiler Construction. In *Proceedings of the ACM SIGCSE-SIGCUE technical symposium on Computer Science and education* (pp. 158–166). New York, NY, USA: ACM.

Sierra, J. L., Fernández-Pampillón, A. M., & Fernández-Valmayor, A. (2008). An environment for supporting active learning in courses on language processing. In *Proceedings of the 13th annual conference on innovation and technology in Computer Science education (ITiCSE '08)* (pp. 128–132).

Urquiza-Fuentes, J., Gallego-Carrillo, M., Gortazar-Bellas, F., & Velazquez-Iturbide, J. A. (2006). Visualizing the symbol table. In *Proceedings of the 11th annual SIGCSE conference on innovation and technology in Computer Science education* (pp. 341). New York, NY, USA: ACM.

Vegdahl, S. R. (2000). Using visualization tools to teach compiler design. In *Proceedings of the second annual CCSC on computing in small colleges northwestern conference* (pp. 72–83). Consortium for Computing Sciences in Colleges, USA.

Waite, W. M., Jarrahian, A., Jackson, M. H. & Diwan, A. (2006). Design and implementation of a modern compiler course. In *Proceedings of the 11th annual SIGCSE conference on innovation and technology in, Computer Science education (ITiCSE'06)* (pp. 18–22).

Werner, M. (2003). A parser project in a programming languages course. *Journal of Computing in Small Colleges, 18*(5), 184–192.

White, E., Sen, R., & Stewart, N. (2005). Hide and show: Using real compiler code for teaching. In *Proceedings of the 36th SIGCSE technical symposium on Computer Science education* (pp. 12–16). New York, NY, USA: ACM.

White, T. M., & Way, T. P. (2006). JFAST: A java finite automata simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer Science education* (pp. 384–388). New York, NY, USA: ACM.

Xu, L., & Fred, G. (2006). Chirp on crickets: Teaching compilers using an embedded robot controller. In *Proceedings of the 37th SIGCSE technical symposium on Computer Science education* (pp. 82–86). New York, NY, USA: ACM.

6.10 Facilitating comprehension of basic concepts in computer language implementation courses

Cita completa:

Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J.L. Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game-based Approach. En SIIE'12: International Symposium on Computers in Education , 1-6. 2012.

Resumen original de la contribución:

Evaluators is an educational tool that lets instructors in computer language implementation courses generate video games from collections of exercises concerning language processing tasks. For this purpose, the tool adopts attribute grammars as a central model of syntax-directed translation and uses a simple metaphor to map attributed syntax trees and semantic evaluation into videogames. These games provide students with an immersive experience which helps them to better comprehend the fundamental concepts behind attribute grammars. This paper describes Evaluators and reports on its assessment with students in a Compiler Construction course at the Complutense University of Madrid (Spain).

Referencia de citas bibliográficas:

[2][5][11][20][24][39][54][56][58][75][109][120][137][158]

Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game-Based Approach

Daniel Rodríguez-Cerezo, Antonio Sarasa-Cabezuelo, Mercedes Gómez-Albarrán, José-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
Madrid, Spain
{drcerezo, asarasa}@fdi.ucm.es, albarran@sip.ucm.es, jlsierra@fdi.ucm.es

***Abstract*— *Evaluators* is an educational tool that lets instructors in computer language implementation courses generate videogames from collections of exercises concerning language processing tasks. For this purpose, the tool adopts *attribute grammars* as a central model of syntax-directed translation and uses a simple metaphor to map attributed syntax trees and semantic evaluation into videogames. These games provide students with an immersive experience which helps them to better comprehend the fundamental concepts behind attribute grammars. This paper describes *Evaluators* and reports on its assessment with students in a Compiler Construction course at the Complutense University of Madrid (Spain).**

Education in Computer Language Implementation, Educational Tool, Game-based Learning, Attribute Grammars

I. INTRODUCTION

Mainstream CS curricula recommendations [1] acknowledge that computer language implementation is a key aspect of CS education, which is addressed to some extend in almost all undergraduate and/or graduate programs on CS.

When teaching this discipline at the Complutense University of Madrid (Spain), we have realized the importance of promoting a clear separation between the specification and the subsequent implementation of the computer language. In order to achieve such a separation, we have adopted syntax-directed translation [2] as a central paradigm for architecting language processors and attribute grammars [12] as a basic model for the specification of syntax-directed translation tasks.

What we have experienced is that the success of this pedagogical strategy strongly depends on how our students previously assimilate the basic concepts regarding attribute grammars, and, in particular, regarding the computational model underlying the formalism. To help our students comprehend this computational model, our first attempt was to use batteries of exercises related to the lower levels of Bloom's taxonomy [4] (in particular, to the comprehension level). Typical exercises consisted of:

- An informal description of a language processing task.
- A formalization of the task using an attribute grammar.
- A sentence in the processed language.

Students had to find a parse tree and explain how to evaluate the semantic attributes of each node.

Although this attempt had a positive effect on the assimilation of basic computational aspects of attribute grammars, this effect was not as good as we would have desired. In particular, we detected an alarming predominance of rote / algorithmic learning. Indeed, most of the students limited themselves to extrapolating, to a greater or lesser extent, our solution templates to the new exercises. Thus, we were aware of the need to develop more effective methods.

Encouraged by our previous experience promoting the active learning of disciplines related to language processing by means of specialized tools [14], we envisioned a simulation environment in which students could resolve the exercises and obtain feedback on their successes and their mistakes. Taking into account the highly motivating component of videogames when applied to teaching and learning [8], we decided to use videogames in which the missions of the student were directly related to the solutions of the language processing exercises. The result was a system called *Evaluators*, which is presented in this article.

Evaluators can be compared to other proposals for the computer-supported teaching and learning of the computer language implementation discipline:

- Some of these proposals are focused on specifying and/or constructing different parts of a language processor (e.g., [6][11]). Therefore, they are of a more advanced nature than *Evaluators*, which is focused on the initial stages of the learning process.
- Other proposals, which are focused on visualizing and/or animating different phases and/or algorithms in language processing [3][7], lack the immersive and motivating features promoted by game-based learning. Additionally, *Evaluators* has a narrower and more specific learning goal: to learn the fundamentals of semantic evaluation in attribute grammars.

(a)

```
expr → expr + term
  expr(1).val = expr(2).val + term.val
  expr(2).ienv = expr(1).ienv
  term.ienv = expr(1).ienv
expr → term
  expr.val = term.val
  term.ienv = expr.ienv
Term → num
  Term.val = toNum(num.lex)
Term → iden
  Term.val = valOf(iden.lex, Term.ienv)
```

(b)

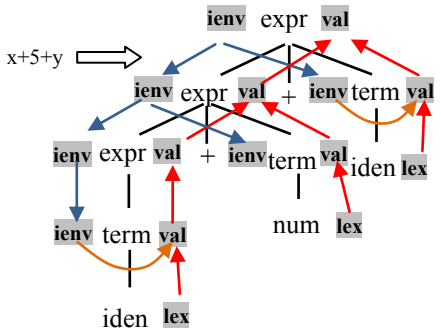


Figure 1. (a) An example of attribute grammar; (b) attributed parse tree associated with a sentence, along with dependencies between attributes.

- Finally, *Evaluators* is similar in spirit to the works reported in [9][10], although it differs from those works in the learning domain: attribute grammars in the case of Evaluators, black-box compilation of Java programs in the case of JV2M [9], and object-oriented design in the VirPlay3D2 environment [10].

The structure of the rest of the paper is as follows: Section II presents *Evaluators*; Section III reports on some results of the tool assessment; and Section IV presents some conclusions and future work.

II. EVALUATORS

This section presents our *Evaluators* system. Given the importance of attribute grammars in *Evaluators*, we will start by briefly summarizing this formalism (subsection II.A). Then we will give a general overview of *Evaluators* (subsection II.B). Next we will describe the metaphor used in *Evaluators* to visualize and interact with attributed parse trees (subsection II.C). Finally, we will examine the game-play in *Evaluators*.

A. Attribute Grammars

Attribute grammars are widely used to model *syntax-directed* translation tasks, in which the processing of the sentences of a formal language is driven by their syntactic structure [2]. Figure 1a shows an example of an attribute grammar concerning the evaluation of a small language that consists of very simple arithmetic expressions.

In order to describe structure, attribute grammars adopt context-free grammars as their basic skeletons. Context-free grammars use syntax rules to define the syntax of the language. Each rule explains the structure of a composite construction (i.e., a *non-terminal symbol*) in terms of simpler composite construction, as well as in terms of primitive construction (i.e., *terminal symbols*). For instance, the rule *exp* → *exp* + *term* in Figure 1a states that "an *expression* may be a simpler *expression*, followed by the + operator, and followed by a term", where the + operator is an example of a terminal symbol. Using syntax rules it is possible to impose a tree-shaped structure on each sentence in the language. This structure is the sentence's *parse tree*, which represents how to apply the syntax rules to generate such a sentence (see Figure 1b for an example). In this tree, internal nodes are labeled with non-terminal symbols, leaf nodes are labeled with terminal symbols, and parent-child relationships are determined by the application of syntax rules.

To add processing capabilities, attribute grammars associate *semantic attributes* with the syntax symbols and *semantic equations* with the syntax rules:

- There are two kinds of attributes: *synthesized* and *inherited*. The value of a synthesized attribute represents the meaning of the symbol (e.g., `val` represents the value of expressions and terms in Figure 1a). The value of an inherited attribute provides additional contextual information (e.g., in Figure 1b, `ienv` represents an environment assigning values to variables). The synthesized attributes of terminal symbols are called *lexical attributes*, and they should be set during lexical analysis (e.g., in Figure 1a, `lex` is a lexical attribute representing the actual number or identifier).
- Semantic equations, in turn, indicate how to compute the values of semantic attributes in syntax rules using the values of other semantic attributes (e.g., the first semantic equation of *exp*→*exp* + *term* in Figure 1a indicates how to compute the value of the sum by adding the values of the operands).

Attribute grammars enable *semantic evaluation* on *attributed parse trees* (i.e., parse trees along with the semantic attributes for each node). Semantic evaluation consists of assigning values to the attributes of each node, and it is driven by the *dependencies* between attributes (see Figure 1b). An attribute depends on those attributes that are required to compute it (in Figure 1b it is indicated by arrows; notice that if

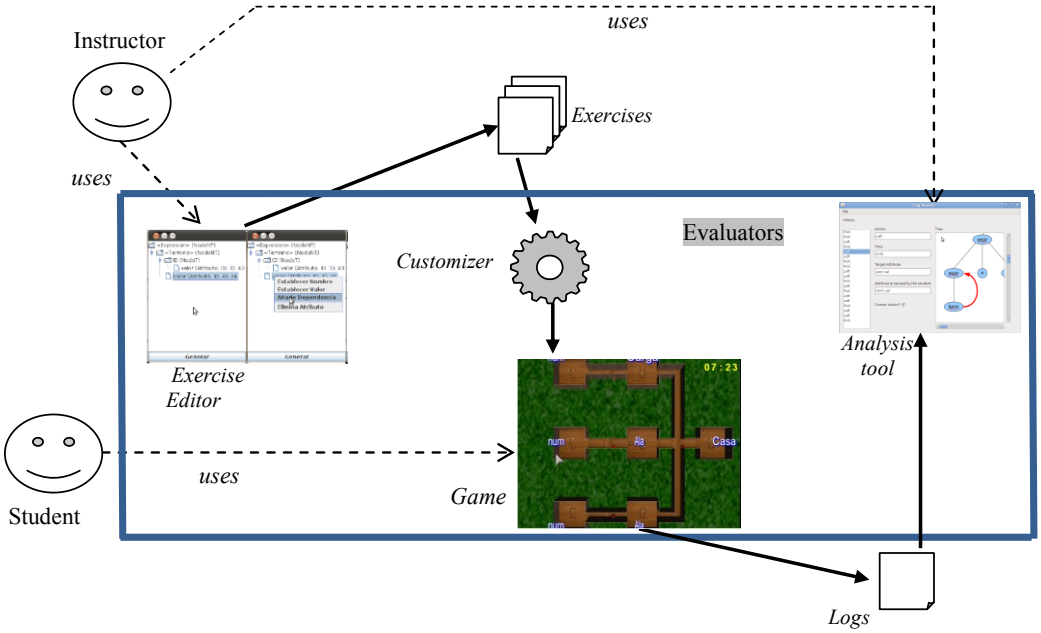


Figure 2. Evaluators workflow.

an arrow goes from an attribute a1 to b1, it means that a1 is used in order to compute b1). Therefore, the only constraint imposed by the attribute grammar on semantic evaluation is that, before computing the value of an attribute, the values of the attributes on which it depends should already have been computed. Aside from this basic constraint, evaluation order does not matter. This insensitivity to evaluation order places attribute grammars at a higher level than other approaches for specifying syntax-directed translation tasks (e.g., *translation schemes*: context-free grammars with interleaved semantic actions that fire during parsing [2]).

In our experience teaching computer language implementation courses, we have realized the strengths of using attribute grammars. They promote the modularity, extensibility and maintainability of the final specifications and processors, and also facilitate the application of systematic implementation techniques. However, we have also realized that the approach is accompanied by a non-negligible learning curve, which is particularly accentuated for students unhabituated to formal and declarative specification methods. Our aim with *Evaluators* was to smooth this learning curve.

B. *An Overview of Evaluators*

Figure 2 summarizes the use of *Evaluators* (see [13] for a more detailed description of the process model followed in the construction of this system). According to this schema:

- First, instructors propose assignments; that is, sets of suitable exercises that students must solve. The structure of these exercises should mirror those that we use in our introductory lessons on attribute grammars (see section I). Therefore, instructors should informally describe the processing task, and provide both an attribute grammar formalizing the task and a sentence to process. In addition, they should provide the solution to each exercise: an attributed syntax tree, along with an explicit representation of dependencies between the attributes (i.e., the *dependency graph* associated to the tree; see example in Figure 1b). While the description of an exercise has a free-text format, the attributed syntax tree should be rigorously encoded. To facilitate this task, *Evaluators* provides an easy-to-use *exercise editor*.
- Once the exercises are available, a *customizer* tool automatically generates a *game*. Each exercise proposed in an assignment represents a level of the game.
- Once the game is generated, students should play it. In order to finish the assignment successfully, students must complete each level of the game (i.e., solve each exercise). The system provides them with the needed feedback on their successes and failures. It also logs students' activity.
- Finally, instructors should inspect the log files by using an *analysis tool*. This way, instructors could detect and analyze the most frequent mistakes in order to determine adequate corrective and reinforcement actions.

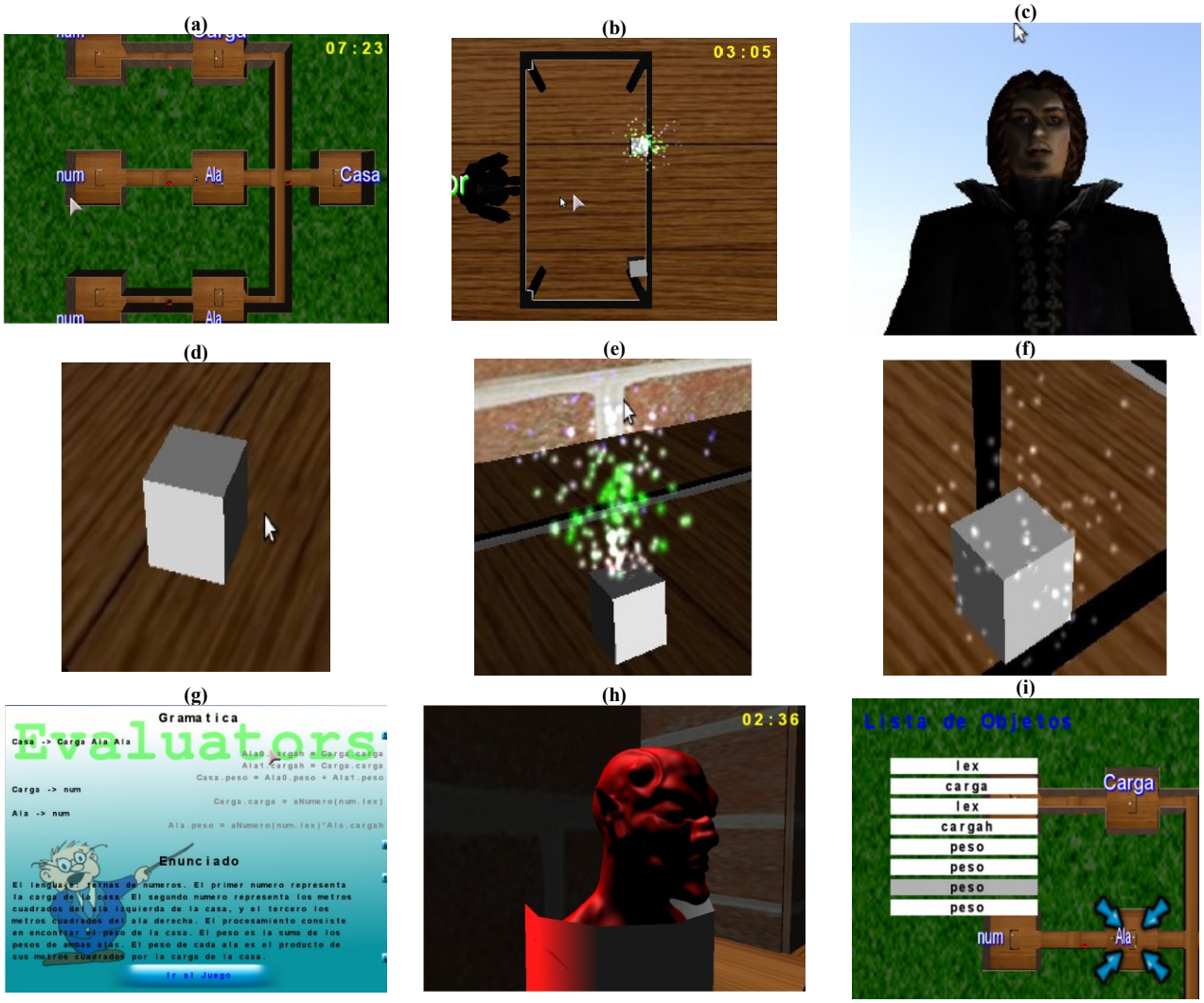


Figure 3. Screenshots of a videogame deployed by Evaluators (see main text for further explanations).

Next subsections go into the details of the videogames deployed by *Evaluators*.

C. *The Evaluators metaphor*

As indicated previously, Evaluators maps attributed syntax tree into videogames. This mapping is based in a metaphor that adopts the following conventions:

- Syntax trees are viewed as maze-like worlds. In these worlds, rooms represent tree nodes, instances of syntax rules are represented as halls, and arcs are represented as corridors (Figure 3a).
- In turn, each room has a table with boxes on it. These boxes represent instances of semantic attributes and can contain objects (the values of the attributes) (Figure 3b)
- Finally, each game includes an avatar which can be commanded by the student. By using the avatar the student can carry objects from box to box, thus reproducing the attribute evaluation process (Figure 3c).

D. *Evaluators game-play*

In *Evaluators*, the student's mission is to complete each level, which involves successfully solving the semantic evaluation problems posed by each exercise. In order to do so, and according to the metaphor described in the previous section, the student avatar should feed to each box appropriate replicas of the objects provided by other boxes. From an attribute grammar perspective, it is equivalent to deciding, for each unknown attribute, the other attributes which are needed to evaluate it. To make this process possible, *Evaluators* provides the student with the following feedback:

- Boxes that correspond to unknown attributes appear as inactive (Figure 3d). However, boxes corresponding to attributes whose values have been calculated emit a beam of multicolor light (Figure 3e).

- Student mistakes correspond to dropping an object in the wrong box (i.e., using the wrong attribute in a computation). In this case, the system shows her a plume of gray smoke (Figure 3f). On the other hand, if the student drops the right object in a box (i.e., she uses the right attribute), the effect will depend upon whether the evaluation process has been completed or not. In the first case, the box will emit the aforementioned multicolor beam; in the second case, the box will remain inactive. Additionally, the student can be forced to restage a level if she exceeds a certain number of errors (the details of this behavior can be also customized by the instructor).

In addition to this basic feedback, the system can provide additional support:

- Names and number of the still-unsolved dependencies of each attribute.
- In addition to the aforementioned beam of multicolor light, the value of the attribute if the instructor introduced it during customization.
- A description of the exercise (Figure 3g), which the student can consult at every point of the game. She can also visualize a global map of the maze (Figure 3a).
- Finally, each hall includes an oracle (Figure 3h). This character can be consulted by the avatar, and provides information on the particular rule associated with the hall.

Lastly, to store the objects (i.e., the values of the attributes), the system includes the concept of an *inventory*. This inventory consists of a list of objects (attribute values). Upon student requirement, using the map of the maze (i.e, in the syntax tree), the system can locate the room (node) where the box (attribute) that originated each object (value) in the inventory resides (Figure 3i).

III. TOOL ASSESSMENT

In this section we report on the experience using *Evaluators* in a Compiler Construction course at Complutense University. In this experiment, 19 students participated in the *Evaluators* usage experience, which was conducted as extra credit. We collected student experiences and favourable / unfavourable attitudes with a survey consisting of structured questions. Table 1 summarizes the results. This qualitative survey consisted of seventeen Likert-scale items, as listed in the left column of Table 1. In order to develop the survey items we were inspired by and adapted the TUP model for evaluating educational software [5]. Survey items were divided into pedagogical and usability aspects. Pedagogical utility and context adequacy, in the first case, and learnability, interaction facilities and overall usability judgment, in the second case, are assessed.

The scale used for student answers is a 4-point scale (Disagree – Slightly Disagree – Slightly Agree – Agree). Table 1 lists the percentage of students who selected "Slightly Agree" or "Agree" with the listed survey items.

Survey results are promising: *Evaluators* has obtained success and popularity within this group of students as a complementary tool for Compiler Construction lectures. Many students consider that it is enjoyable and useful for understanding the different aspects related to attribute grammars (i.e., semantic attributes, semantic equations, processing). Therefore, the tool's objectives have been achieved to a great extent. It is worthwhile to point out that students find *Evaluators* interesting in order to realize the differences among the different types (i.e., lexical, synthesized, inherited) of attributes, although we did not anticipate this consequence when developing the tool. As a result of these positive aspects, a high number of students would recommend *Evaluators* to their classmates.

TABLE I. AGREEMENT WITH SURVEY ITEMS

Survey item	% agree
Pedagogical utility of the tool	
It helps me understand the attribute grammar processing	73,7
It helps me understand the role of semantic attributes	73,7
It helps me understand the role of semantic equations	68,4
It helps me realize the differences among the different types of grammar attributes	73,7
I do not find it hard to realize my own mistakes when using the tool	36,8
Context adequacy	
Evaluators suitably complements lectures	84,2
Evaluators motivates me to study attribute grammar processing	52,6
Learnability	
I can rapidly start working with the tool without a long period of training	63,2
Mazes are a straightforward metaphor for representing attributed syntax trees	73,7
Using tables to collect information about attributes seems natural to me	52,6
Interaction	
I have used the problem statement several times	68,4
I have used the oracles	10,5
It is easy to identify which attribute instance corresponds to each attribute copy in the inventory	31,6
It is always easy to know the state of problem solving	73,7
Overall usability judgment	
Evaluators is easy to use	42,1
I enjoyed using Evaluators	73,7
I would recommend Evaluators to other Compiler Construction students	68,4

We are also aware of some aspects of the tool that need to be improved. The tool does not show a very steep learning curve, but some difficulties of use remain. Clearly, oracle functionality should be improved. Finally, clarifying which attribute instance corresponds to each attribute copy in the inventory would reduce student mistakes. We are currently working on solving these shortcomings, as stated in the next section.

IV. Conclusions and Future Work

This paper has presented *Evaluators*, a system that enables a game-based approach to learning the fundamental concepts of attribute grammars. The system lets instructors generate multilevel videogames from collections of exercises authored by the instructors. Using these games, students can reinforce their comprehension of the basic computational mechanisms behind attribute grammars. We have also presented some assessment results concerning the tool.

Currently we are working on solving the deficiencies of the tool detected during its assessment. In particular, we are thinking up more straightforward metaphors for representing the attribute information. Also, we are considering the inclusion of additional feedback associated with the types of student mistakes, which clearly involves an extra authoring effort. In order to better deal with tool improvements, we plan to use a new survey to collect student suggestions about the items most poorly assessed in the usage experience presented in this paper. The new survey will include an open-ended (free form) question for each aspect surveyed. Additionally, the analysis of log files could also shed some light. In the short term, we plan to formatively assess *Evaluators* with a group of instructors, as well as to assess the impact of *Evaluators'* use on student performance. Student and instructor feedback will be used to improve the tool.

Acknowledgments

This research was partially supported by project grants TIN2010-21288-C02-01, TIN2009-13692-C03-03, and Santander-UCM GR 42/10, group reference 962022. Also, it is supported by the Spanish University Teacher Training Program (EDU/3445/2011). We also would like to thanks Rafael Fernández-López and Ángel Valero-Picazo by their invaluable contribution to the implementation of a previous version of *Evaluators*.

References

[1] ACM/IEEE. Computer Science Curriculum 2008: An Interim Revision of CS 2001. 2008

[2] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques and tools (second edition), Addison-Wesley, 2007

[3] Almeida-Martínez, F.J., Urquiza-Fuentes, J., Velázquez-Iturbide, A. Visualization of Syntax Trees for Language Processing Courses. J. UCS Journal of Universal Computer Science 15(7): 1546-1561 (2009).

DOI = http://dx.doi.org/10.3217/jucs-015-07-1546

[4] B. Bloom, E. Furst, W. Hill, and D.R. Krathwohl: Taxonomy of Educational Objectives: Handbook I, The Cognitive Domain, Addison-Wesley, 1956.

[5] Bednarik, R., Gerdt, P., Miraftabi, R., Tukiainen, M.: Development of the TUP Model - Evaluating Educational Software. ICALT'04 Proceedings of the 4th IEEE Int. Conference on Advanced Learning Technologies: 699-701

[6] Demaille, A., Levillain, R., Perrot, Benoît. A set of tools to teach compiler construction. ITiCSE '08 Proceedings of the 13th annual conference on Innovation and technology in computer science education: 68-72.

DOI = http://dx.doi.org/10.1145/1597849.1384291

[7] García-Osorio, C., Gómez-Palacios, C., García-Pedrajas, N. A tool for teaching LL and LR parsing algorithms. ITiCSE '08 Proceedings of the 13th annual conference on Innovation and technology in computer science education: 317.

DOI=http://dx.doi.org/10.1145/1597849.1384360

[8] Garris, R., R. Ahlers, and J.E. Driskell, Games, Motivation and Learning: A Research and Practice Model. Simulation & Gaming, 2002. 33(4): p. 441-467.

[9] Gómez-Martín, M-A., Gómez-Martín, P-P., González-Calero, P-A. Game-Driven Intelligent Tutoring Systems. ICEC'04 Proceedings of the Int. Conference on Entertainment Computing: 21-39

[10] Jiménez-Díaz, G., Gómez-Albarrán, M., González-Calero, P-A. Pass the Ball: Game-Based Learning of Software Design. ICEC'07 Proceedings of the Int. Conference on Entertainment Computing: 49-54

[11] Mernik, M., Žumer, V. 2003. An Educational Tool for Teaching Compiler Construction. IEEE Transactions on Education, 46, 1, 61–68.

DOI=http://dx.doi.org/10.1109/TE.2002.808277

[12] Paakki, J. 1995. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computer Surveys, 27, 2, 196-255.

DOI=http://doi.acm.org/10.1145/210376.197409

[13] Rodriguez-Cerezo, D., Gómez-Albarrán, M., Sierra, J.L. From Collection of Exercises to Educational Games: A Process Model and a Case Study. ICALT'11 Proceedings of the 11th IEEE International Conference on Advanced Learning Technologies: 282-284.

DOI= http://dx.doi.org/10.1109/ICALT.2011.187

[14] Sierra, J.L., Fernández-Pampillón, A.M., Fernández-Valmayor. A. An environment for supporting active learning in courses on language processing. ITiCSE '08 Proceedings of the 13th annual conference on Innovation and technology in computer science education: 128-132.

DOI = http://dx.doi.org/10.1145/1597849.1384307

Capítulo 7: Versión resumida en Inglés / Summary in English

EDUCATIONAL TOOLS TO FACILITATE THE ADOPTION OF SOFTWARE LANGUAGE ENGINEERING BY DEVELOPERS

About this document

This PhD dissertation is presented as a collection of journal and conference publications. The papers presented are the following:

- Rodríguez-Cerezo D., Sierra, J.L. Introducing a Design-Preserving Implementation Strategy in a Compiler Construction Course. In SIIE'13: Actas del XV Simposio Internacional de Informática Educativa, 24-29. 2013.
- Rodríguez-Cerezo D., Sarasa-Cabezuelo A., Gómez-Albarrán M., Sierra-Rodríguez J.L. User-Centered Development of Generative Educational Systems for Computer Engineering: The Evaluators Case Study. International Journal of Engineering Education, 31(3): 751–763. 2015. (ISI WoK JCR IF, 2014: 0.582).

- Rodríguez-Cerezo, D., Gómez-Albarrán, M., Sierra, J. L. A Process Model for the Generative Production of Interactive Simulations in Engineering Education. In TEEM'13: Proceedings of the First International Conference on Technological Ecosystem for Enhancing Multiculturality, 95-103. 2013.
- Rodriguez-Cerezo, D., Gomez-Albarrán, M., Sierra, J. L. From Collections of Exercises to Educational Games: A Process Model and a Case Study. In ICALT'11: Proceedings of the 11th IEEE International Conference on Advanced Learning Technologies, 282-284. 2011. (Conference classified as CORE B).
- Rodriguez-Cerezo, D., Gómez-Albarrán, M., Sierra-Rodríguez, J. L. Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts. In ITiCSE'13: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, 28-33. 2013. (Conference classified as CORE A).
- Rodriguez-Cerezo, D., Henriques, P. R., Sierra, J. L. Attribute Grammars Made Easier: EvDebugger a Visual Debugger for Attribute Grammars. In SIIE'14: Proceedings of International Symposium on Computers in Education, 23-28. 2014.
- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools. Computer Science and Information Systems, 9(3): 983-1017. 2012. (ISI WoK JCR IF, 2012: 0.549).
- Rodriguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. Implementing Attribute Grammars Using Conventional Compiler Construction Tools. In FedCSIS'11: Proceedings of the Federated Conference on Computer Science, 855-862. 2011.
- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J. L. Serious Games in Tertiary Education: A Case Study Concerning the Comprehension of Basic Concepts in Computer Language Implementation Courses. Computers in Human Behavior, 31: 558-570. 2014. (ISI WoK JCR IF, 2014: 2.694).
- Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J.L. Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game-based Approach. In SIIE'12: Proceedings of the International Symposium on Computers in Education , 1-6. 2012.

7.1 Introduction

7.1.1 Motivation

The concepts and techniques implied in the design and implementation of computer languages are an essential part in the basic training of any computer science engineer. In fact, these concepts are included in the main curricula recommendations for university studies in Computer Science [2]. Besides, these

concepts are essential to understand advanced development approaches such as generative ones [36] [83], model-driven development techniques, development methods driven by domain specific languages (DSLs) [50] [88] [171], etc. Thus, most teaching plans of university studies of Computer Science include subjects that address these topics.

Historically these contents are addressed by introductory courses to compiler construction techniques [5]. Those courses outline development models strongly supported by formal specification models, as well as systematic specification-driven implementation techniques. Indeed:

- In those models, the development begins by specifying the different aspects of a computer language, and its processing using specific formalisms for each aspect (regular expressions, context-free grammars, attribute grammars, etc.).
- Then, systematic implementation techniques (usually supported by specific purpose tools) are applied to build the language processors.

In this manner, the language processors obtained are highly dependent on the quality of the specifications, as well as the ability of the developer to transform those specifications into operational implementations. However, the students of this subject usually despise the importance of the specification formalisms, because they find these formalisms excessively abstract and disconnected from other conventional development approaches. Therefore they do not reach the comprehension and dexterity required to carry out the practical application of the specification techniques. The result is the confection of incomplete or incorrect specifications, an early transition to more satisfying implementation phases, an *ad hoc* development of language processors with a disconnection between the implementation and the specification, and, as a result, low quality language processors that have multiple malfunctions or work correctly but are very difficult to trace, maintain and extend [158] [159].

From these problems is where arises the motivation to carry out this thesis, which pretends to provide educational tools and methods to instructors and students of subjects related to the mentioned introductory compiler construction courses, in order to facilitate the acquisition of basic knowledge and dexterities required during the specification and systematic implementation of language processors driven by the specification phases.

This thesis has been developed within the research group ILSA (Implementation of Language-Driven Software and Applications) at the Complutense University of Madrid. Next, the research goals and the research proposal are exposed.

7.1.2 Research goals and research proposal

As described above, the right development of a language processor depends on a correct orchestration of formal specification methods, and the subsequent application of systematic development techniques driven by the specifications. However, the students of compiler construction courses have, at a greater o lesser extent, problems to assimilate properly this orchestration [158] [159]. Thus, the main objective of this thesis is to provide educational methods, strategies and software tools in order to facilitate the correct assimilation of the mentioned

orchestration, essential to correctly achieve the systematic development of language processors. These methods and tools take advantage of the features offered by the new trends in education and eLearning to present these concepts in a more closer, accessible and attractive way to the students.

In this manner, the research posed in this thesis work is based on the development of educational strategies and tools to propitiate the comprehension of the specification methods and their correct use in the development of language processors, as well as to empirically validate their suitability through different experiments performed in courses related to the mentioned development. For this purpose the following goals have been set:

- The first goal is centered on the development of a teaching method to improve the comprehension of the basic concepts involved in the specification of a language processor, on the implementation of this method by means of an appropriate set of software tools, and on the assessment of the pedagogical suitability of both the educational method and the tools.
- The second goal is centered on proportionating a development strategy that facilitates to the students the early development of a language processor from its specification, as well as on assessing the educational impact of the strategy.
- Finally, the third goal is centered on providing a development environment of language processors based on the specification that integrates the comprehension aspects of the specification as long as the aspects related to the development of the processors from these specifications. Also, it contemplates the assessment of the development environment from an educational point of view.

To address the first goal, the formalism of the attribute grammars [89] [90] [120] has been chosen as the basic formalism to specify the language processors. This choice is justified, on one side, by the wide use of the formalism in many courses of introduction to compiler construction [56] [100]. On the other side, it is justified because this is the formalism used in the courses where the proposals have been empirically validated. In this manner, the works related with the achievement of this objective have been focused on the *comprehension* aspects of the formalism chosen, the attribute grammars, in order to help students to understand the basic concepts and mechanism of semantic evaluation that underlay the formalism. Thus, the aim has been to improve the assimilation of these concepts and mechanisms in the early stages of the learning-teaching process. The hypothesis here has been that this early assimilation will contribute to help students to better use the formalism to satisfactorily overcome basic specification tasks of syntax-directed translation in which the language processor construction relies, and, as a result, to improve the global language processor development process. In order to accomplish this objective, a learning-teaching strategy centered in the evaluation process of the attribute grammars has been designed, together with different software educational systems based on this strategy. The common denominator of these systems is to promote a *generative* and *problem-driven* approach involving, in a coordinated manner, teachers who pose semantic evaluation exercises to solve, as well as students who solve these exercises. Likewise, and although the work is completely focused on attribute

grammars, a major effort to abstract the process of building such systems has been also invested, so that the method can easily be extrapolated to treat other specification formalisms.

The second goal has been addressed through the creation of a language processor development strategy supported by conventional language translators tools (CUP, JavaCC, etc.) that lets students easily implement their specifications based on attribute grammars via typical translation models. In this manner, the work on this objective has been focused more on improving aspects of effective creation of the processors by students, promoting, to this end, a systematic process that elicits the direct encoding of the formal specification in the tool. The result is a first executable prototype of the language processor implemented through a conventional language processor generation tool, which faithfully preserves the specification. This prototype can serve as a basis for systematic improvements and other changes oriented to adding efficiency to the resulting processor.

The third objective serves, eventually, as a nexus to the first two. To this end, a development environment for the generation of language processors has been implemented that is able to turn attribute grammar based specifications of these processors into working implementations supported by conventional generation tools. For this purpose, the environment applies the hand-coding implementation patterns designed for the objective two. Besides, in order to facilitate the comprehension of the designs and to offer support for debugging them, the environment offers a sophisticated visual debugger, based on the educational strategy developed during the achievement of the objective one.

Likewise, and as stated in the goals, we will make an special emphasis on the empirical evaluation of each of the strategies and tools created. For that, different studies and experiences among teachers and students of subjects related to the construction of Language Processors in both the Complutense University of Madrid and the University of Minho (Portugal) have been conducted. Based on the results of each of these experiences, it has been possible to verify the assessment of the various proposals by students and teachers, the perceived usefulness of each of the artifacts conceived, and their educational effectiveness.

7.2 State of the art

As already indicated in the previous section, this thesis deals with the improvement of the teaching-learning process in subjects related to the design and implementation of computer languages regarding specification aspects (in particular those relating to the use of declarative formalisms for the description of syntax-directed language processing tasks) and systematic implementation of processors from specifications. Thus, in this section the most important elements to face in order to carry out the contextualization of this thesis work are reviewed.

The section begins by reviewing the most relevant aspects to this thesis related to teaching approaches concerning language processor development subjects (subsection 7.2.1). Then, given the importance that both simulations and educational games acquire in this thesis regarding the understanding of the basic concepts involved in the specification of a language processor, we review the main features of these simulations and educational games, with special emphasis on the

teaching of computer science in general, and the subjects related with Language Processors in particular (subsections 7.2.2 and 7.2.3). Subsection 7.2.4 reviews the most relevant aspects of language processors development tools, as well as the educational uses of these tools. Finally, subsection 7.2.5 concludes this analysis of the state of the art, showing the most relevant conclusions for the development of the thesis.

7.2.1 Teaching the Language Processors subject

The main objective of this thesis is to improve teaching-learning aspects of specification and systematic implementation of language processors. The aim is to alleviate the problem that, historically, is surrounding the teaching-learning of subjects related to the design of computer languages and the implementation of their processors:

- Courses in building language processors typically contemplate the coordinated use of a large amount of theoretical concepts and systematic development techniques. Teachers can not cover all in a complete way, making it difficult for students their application to designing and actual developing a processor [4].
- Students also must successfully complete the development of a processor for a computer language, activity in which they find most of the problems [66].
- Students are not able to connect the theoretical concepts and formalisms explained in class, with the technical concepts and development methodologies. It causes students to develop processors that do not work properly, often due to some erroneous specifications [159].
- Students consider most of the concepts taught in the course as too complex, because of its formal nature. It causes a loss of motivation, as they do not consider those useful concepts to their daily work practice [174], despite being fundamental for new (model-driven and/or language-driven) development approaches [50] [88].

To address the problems associated with these subjects, teachers apply different teaching strategies to facilitate students the assimilation of the concepts involved in them. These strategies place particular emphasis on the practical part of the teaching-learning as an essential component for successful assimilation of the subject. Three of the most representative strategies are detailed in the following subsections: strategy based on the development of a language processor for complete computer language, strategy focused on building small languages processors oriented to illustrate different processing patterns, and strategy based on the analysis and debugging of a real processor. Finally, the section concludes with a discussion of these strategies.

7.2.1.1 Strategy based on the development of a processor for a complete language

This strategy is the most commonly used by many teachers. The strategy proposes to the students to develop a processor for a complete computer language that translates source code into a suitable object language. In this regard:

- Many approaches are based on choosing programming languages as source languages. For this purpose, a wide range of prototypical reduced programming languages (COOL [6] or MINIML [19], for example) are available. These languages typically share the following features: inclusion of a few basic data types, a set of basic operations for those types, control structures, data types defined by the programmer and an abstraction mechanism.
- Other approaches focus on less conventional languages but have an extra attraction for students due to their purpose. Examples of these languages are languages for graph representation [177], for simple figure drawing [145] or robot programming languages [181]. Their main advantage is its attractive orientation, which can enhance student motivation.

Regardless of the language chosen by the teacher, when implementing the language processor they may also contemplate different options: requiring students to implement the processor from scratch [48] [153], providing components developed to handle different aspects of language processing [18] as a starting point and help for students during development, and allowing students to use conventional tools (YACC [154] Bison [97] or JavaCC [91]) for the development of processors as shown in [39] [102].

7.2.1.2 Strategy based on small projects of language processing

The main problem of the strategy based on the development of a processor for a complete language is that, although the language supported can be simple, the implementation of the processor may not be trivial. Indeed, the students can found problems during any specification or development phase that could delay the implementation process (or even cause the project to fail). In addition, there is a mismatch between the pace of the lectures and the project, which limits the time students have to finish the project.

Given these shortcomings, the strategy based on small projects of language processing has been proposed. In this strategy, as the lessons progress during the course, teachers propose small projects concerning simple languages directly related to the concepts revised in class. In addition, teachers often provide students with parts of the processors implemented to focus the attention of the students on the concepts related to the project, which should have been also introduced in the classroom. In [95] [157] the benefits of this strategy are analyzed.

7.2.1.3 Strategy based on the analysis and debugging of a real language processor

This strategy arises in response to the main problem exhibited by project-based strategies: these projects are not enough to teach all the techniques and concepts involved in the development of processors for real computer languages. Therefore, in [108], the debugging of a professional processor is proposed to illustrate the concepts and techniques implied in the actual development of real-world language processors. Lessons are organized in laboratory sessions where teachers use breakpoints to display the most significant variables involved in the processing. Thus, teachers show how the techniques and methods taught in the lectures are applied. During these sessions, teachers do not completely debug all the internal processes, but focus on the most important and immediately related concepts taught in class.

7.2.1.4 Discussion

We have exposed different methodologies for teaching Language Processors that adopt different pedagogical approaches, showing different advantages and disadvantages. In particular:

- The implementation of a processor for a complete language requires students to develop a processor for a computer language using the techniques and formalisms introduced in lectures. On the other hand, this kind of project gives students an opportunity to apply their skills in project management and software engineering, due to the magnitude of the project and the required quality. However, the main disadvantage of this approach is that the complexity of the development can exceed the skills in design and implementation of computer languages of any average novel student.
- The methodology based on the accomplishment of small language processing projects has as its main strong point that, since the projects are proposed immediately after the related lectures are delivered, students are able to more easily retain the concepts taught. However, these small projects make students lose the more global perspective provided by the addressing of a major project.
- Finally, the analysis and debugging of a real language processor let teachers show how the concepts and methods discussed theoretically in the classroom are applied in the real world, by mean of the debugging of an actual language processor. However, these professional processors are highly optimized and sometimes it is difficult to identify the techniques and algorithms presented in classroom within the debugged code. In addition, students become mere spectators, unable to properly develop the skills required for specification and implementation.

Regardless of their advantages and disadvantages, these three strategies have a common denominator: they primarily focus on issues relating to implementation. This strong focus on implementation issues, in detriment of the specification aspects, causes students to find difficulties understanding the basis of formal specifications shown in class, and which are necessary for the proper development of the proposed processors in the methodologies described.

7.2.2 Educational simulations

Based on the importance that specification aspects acquire in subjects related to language processors, in this thesis it is considered that the improvement in the understanding of those aspects is essential in order to improve the overall teaching-learning process of those subjects. To do this so, this thesis promotes the use of different types of interactive simulators to facilitate comprehension of the basic concepts of formal specifications in the field of design and implementation of computer languages (attribute grammars, particularly). That is why, in this section, the most relevant aspects of such simulations are reviewed.

Educational simulations facilitate the understanding of concepts by students and help motivate them, stimulate their intellectual curiosity, a sense of personal control, and perseverance [21] [79]. Widely used in the teaching of different fields, such as aviation [161] or the medicine [103], the simulations show to students interactive virtual environments where they can learn through experimentation and interaction with the environment presented. Normally, designers of these simulators create virtual environments reflecting the needs of students. This environments will direct and focus the educational paths followed by students, who will use the simulators. Still, the virtual simulation environment need not necessarily be based on real environments: it is the case of those simulators focused on theoretical models. That is why simulators are useful to teach the operation of various algorithms and other computer-related concepts.

In the following subsections, we describe different simulators based on visualization to enhance the learning of different subjects involved in Computer Education (subsection 7.2.2.1), in particular, those oriented to subjects related to the design and implementation of computer languages (subsection 7.2.2.2). Subsection 7.2.2.3 presents a discussion of the most relevant aspects of these approaches to the development of this thesis.

7.2.2.1 Simulation in Computer Science

Being aware of the power of simulation as a tool to improve student learning, many teachers make use of educational simulations. In particular, teachers of Computer Science are more aware of these skills in presenting the performance of different algorithms and formalisms, and also are able to develop their own applications and offer them to other teachers.

Representative examples of such tools in the field of higher education in Computer Science are *Query Simulator System* [10], a tool based on visualizations developed for the data base field, *Java Execution Simulator (JES)* [131], a simulator for the execution of Java code oriented to the early stages of object-oriented programming courses, *UNIX Concurrent I/O Simulator* [132], a C code simulator that represents concurrent file input/output through visualizations, *Address Translation Simulator* [133], a simulation tool for the internal performance of operating system address translation, or *Animation of Linked Lists in Java* [40], a Java-based software library oriented to replace predefined Java linked lists in order to provide programs with the capability of visualizing this structure.

The main feature of all these simulators, as well as other simulators of similar nature [30] [169], is that the concepts are presented in an attractive way, allowing students to discover and learn these concepts through experimentation and exploration. Normally, the interaction offered varies according to the teaching concepts and the degree of involvement that is expected from the student: from simulators as *JES* or *Query System Simulator* that allow students to observe the operation of Java programs or SQL queries that they develop by themselves, to *Address Translator Simulator*, which emulates step by step the behavior of each components involved in the process of address translation. It is also possible to highlight how these simulators show the information using visualizations and animations. Similarly, two recurring types of representations used in these simulations can be identified: table-based representations, and abstract representations of the components involved in the particular simulation.

7.2.2.2 Simulation for the teaching of Language Processors

Language processors subjects are also likely to benefit from educational simulations. Thus, this section describes different software tools based on simulations designed to improve the teaching-learning of these subjects. For this purpose, these tools are categorized into three different groups: *theoretical machines simulators* (subsection 7.2.2.2.1), *simulators of analysis algorithms* (subsection 7.2.2.2.2) and *other aspects of processing simulators* (subsection 7.2.2.2.3).

7.2.2.2.1 Theoretical machines simulators

These simulators are born from the need to present the algorithms and theoretical machines involved in language processing in a dynamic, attractive and interactive way. Normally, these concepts are presented to students using the blackboard or slides, and often students are unable to understand the inner behavior of the artifacts presented, because the explanations usually are tedious and difficult to follow. However, thanks to these simulators, students can interact with these machines and learn in a more experimental and attractive way. Some examples of this type of simulator are *jFAST* [179], a simulator capable of showing the performance of different automata and theoretical machines (like finite automata, stack automata, Turing machines, etc.) through an attractive and accessible interface for students, and *Simulator of Finite State Automata* [64] [65], a similar tool to jFAST which lets students create and simulate different types of automata and theoretical machines.

Although the usefulness and appropriateness of these tools and others alike, as [134], for teaching-learning concepts related to the construction of language processors subjects is undeniable, these tools are focused on the theoretical models of artifacts relating to the implementation (e.g., automata), instead on the specification assets preceding such devices (e.g., regular expressions, grammars, etc.).

7.2.2.2.2 Simulators of analysis algorithms

This set of tools enables to view the operation of various types of syntactic-semantic analysis algorithms. In this way, students are able to understand how these algorithms work thanks to the simulation of the different structures involved in their internal processes. Students can emulate analysis algorithms and observe how structures vary as they progress. Some representative examples of such tools are BURGRAM [54], a simulator based on visualization oriented to show the behavior of different top-down and bottom-up parsing algorithms, SEFALAS [77], a simulator centered in the teaching of different scanning and parsing algorithms, and CUPV [81], a tool for the specification of language processor based on CUP [72] that emulates the dynamics of the parse stack.

The simulators discussed throughout this subsection are focused on providing additional information to language processor generated with language development tools (CUP, ANTLR, etc.) so students are able to understand the internal process of parsing supported by these tools. Even, some of these simulators are used as highly specialized debuggers for students during the practical sessions, and for the development of language processors requested by teachers. As a counterpoint, these tools are focused exclusively on showing the internal operation of the language processors implemented, leaving aside aspects related with the preceding specification.

7.2.2.2.3 Simulators of other aspects of processing

Besides simulators of analysis processes, it is possible to address the simulation of other aspects of the processing. Thus, this type of software shows the construction, maintenance and querying of data structures implied in the language processors, for example: the symbol table, the parse tree, or the execution of resulting object code. The following are three representative examples of such systems are: *SOTA* [165], a simulator able to show the symbol table of a language processor and the actions performed on it, *Tree-Viewer Library* [168], a Java software library that help students to understand their own processors through the construction of syntax tree visualizations, and *Annotating Debugger* [168], a stack virtual machine debugger able to execute machine code generated by a compiler that visually displays the behavior of this machine.

The educational tools studied above, along with other similar (e.g., VAST to visualize the process of building parse trees [11] [12]) provide students with a more attractive way to explore different data structures involved in language processing and their role during the process. To do it so, they make use of simulation based on visualization where the data flow in each of these structures is showed. Their educational value has been demonstrated in different studies, but again these systems are mainly focused on issues related to the implementation aspects, rather than on the previous aspects of specification on which the implementation is based.

7.2.2.3 Discussion

All the tools presented in previous sections are intended to enhance the educational experience in various areas related to Computer Science, and specifically with subjects related to the construction of language processors. These tools make extensive use of the pedagogical power of interactive visualizations to present the concepts [71] [73] [166], and they raise as powerful instruments to improve the teaching of certain concepts related to computer science in general, and in subjects related to language processors in particular. We have observed many significant experiences that can be drawn from good practices. In particular, in relation to the tools focused on subjects related to language processors, we have analyzed different types according to the concepts they covered. Regardless of their field, all tools provide to students a more attractive way to internalize both theoretical concepts and the operation of various stages during language processing.

However, despite the many advantages of simulation-based approaches, it is important to point to the fact that simulators related to the Language Processors course described above are primarily oriented to teaching processes and concepts related to the implementation of processors. However, aspects of specification do not receive attention. This is obvious to simulators related to analysis algorithms and other aspects of processing, as they focus on teaching structures and internal processes of the final implementation. It is also true for theoretical machines simulator, because although these may seem closer to specification aspects, these machines are abstract models of devices used in the implementation of language processors. Therefore, these approaches do not help to improve the understanding and use of the specification formalism by students, but to better understand how the final implementations work to a higher abstract level.

7.2.3 Educational games

This thesis also promotes the use of serious games as mechanisms to enable the understanding of basic concepts related to the specification of language processors based on attribute grammars. Thus, this section will review the most relevant aspects for this thesis about educational games, a type of artifact that has been introduced into the world of eLearning and educational software [13] [56] [57].

The idea of using games to teach is not new: they have already been used for years to teach different concepts and to help acquiring skills, especially in the childhood [75]. Thus, looking for educational uses of video games is something completely natural. In particular, video games can allow the user to learn and experiment with processes and techniques very close to real life as they can immerse the user into virtual worlds designed for such educational purposes [156]. Educational video games have four fundamental characteristics that position them as a valuable educational tool [56]: videogames are fun, immersive, stimulate the cooperation and competitiveness, and encourage the creation of user communities.

As a consequence, educational games are positioned, potentially, as a powerful educational tool. Currently, educational games are used in various fields and

subjects. In particular, developers and designers have created interesting proposals to facilitate the teaching and learning process in the field of Computer Science. These proposals are described in the following sections: subsection 7.2.3.1, and subsection 7.2.3.2 in relation to the particular field of Language Processors. Subsection 7.2.3.3 discusses the most important issues concerning the development of the thesis.

7.2.3.1 Use of educational games in Computer Science

This section discusses the use of educational games oriented to the teaching and learning of Computer Science subjects. These games are designed to improve student learning in different aspects of Computer Science subjects, such as algorithms, project management and computer architecture. Some representative examples are: *Age of Computers* [116], an educational videogame focused on the teaching of *Computers Fundamentals* subjects, *Wu's Castle* [45], a serious game oriented to teach loop operations as contemplated in first programing courses, *Input/Output on Nintendo®DS* [94], which proposes the use of this game console to teach input/output concepts in computers, and *SimSE* [117], a videogame to simulate the role of a development team manager for students of Computer Science subjects.

The games presented above are oriented to teaching different subjects related to Computer Science. They are able to teach students, in an interesting and attractive way, relevant concepts for which they were designed, using metaphors to adequately transform these concepts into video games with educational value (for example, Wu's Castle). They also have the ability to provide students, immediately, information about failures (or successes) that they could accomplish, to guide their learning. On the other hand, some of the examples discussed have features that enhance the collaborative learning (for example, Age of Empires) or allow teachers to assess the progress of students (e.g., SIMSE). In conclusion, these examples are representative precedents of using educational games to teach concepts related to Computer Science, and illustrate the benefits and features of this type of educational software.

7.2.3.2 Use of educational games on Language Processors

As for the matters related to the construction of language processors, with respect to educational games, there is not a variety of these that specifically cover the concepts taught in this subject. Perhaps one of the exceptions is JV^{2}M, an educational game-based tool that is oriented to teach the operation of the Java virtual machine (JV^{2}M) and the translation of programs written in Java into JV^{2}M *bytecode* [33]. The main goal is that students understand the translation of Java *bytecode* programs by understanding the internal implementation of these programs.

Is worth mentioning that this educational game has obtained good results among students [58]. It is also important to highlight the lack of educational games aimed at teaching subjects related to the design and implementation of computer languages.

7.2.3.3 Discussion

During this section we have described the different characteristics and qualities of video games that make them a powerful educational tool, primarily due to possessing attractive component and inviting students to use, and learn with, them. This also justifies its use in various fields related to Computer Science.

However, it is important to note the small number of educational games oriented specifically to the teaching-learning of subjects related to the construction of language processors. This is probably because the concepts and techniques involved in subjects related to the design and implementation of computer languages do not lend themselves easily to be taught using serious games due to its abstract nature. Ultimately, it may be difficult to find appropriate metaphors for these matters, and which in turn result entertaining and attractive for students. These metaphors determine how the concepts and processes that are to be treated in a serious game will be represented by an attractive mechanical game-play and, at the same time, suitable to the students, and their educational needs.

Notwithstanding the lack of serious games applied to the field of teaching-learning of language processors, it is interesting to highlight the example discussed in the previous section, JV^{2}M, illustrating the operation of the translation of Java code to *bytecode* of Java virtual machine. While JV^{2}M clearly illustrates the feasibility of using serious games on this type of subjects, the system is mainly focused on the implementation aspects of the translation process and the implementation of programs as virtual machine code, without addressing the issues related to the previous specification and the underlying fundamentals.

7.2.4 Language processor development tools

As mentioned in subsection 7.2.1, the typical courses on subjects related to the design and implementation of language processors are divided into two main parts: first, the teaching of theoretical concepts and techniques that compound the subject, and, in the other hand, the application of these for the development of a processor. In connection with the practical side of development, there are two possibilities, both widely used:

- To make development directly in a general purpose programming language. This alternative is feasible for small languages and simple implementation techniques (for example, predictive-recursive translators [5]). However, the approach is no longer practicable for more complex or more elaborate implementation techniques.
- Using specific software tools to develop language processors. These tools generally operate from high-level specifications of different aspects of the language to process and its processor itself. Apart from facilitating the development, these tools are especially attractive from the point of view of education, making emphasis on aspects of specification rather than on implementation issues.

Thus, in this thesis the development tools of language processors play a central role, as long as they are able to accept as input different types of specifications of language processors. This section describes the most relevant aspects of these

development tools for the thesis. Subsection 7.2.4.1 discusses the language processor generation tools based on the formalism of attribute grammars. Subsection 7.2.4.2 discusses the more conventional generation tools based on translation schemata. Subsection 7.2.4.3 discusses how to use these tools based on translation schemata to implement specifications based on attribute grammars. Subsection 7.2.4.4 describes the educational use of the language processor generation tools. Subsection 7.2.4.5 discusses, finally, the most important concerns of the different aspects covered in the section.

7.2.4.1 Tools based on attribute grammars

Tools based on attribute grammars accept as input attribute grammars specifications and produce, as a result, the specified language processors.

Attribute grammars are widely used to model syntax directed translation tasks, in which the processing of the sentences of a formal language is driven by their syntactic structure [89] [90] [120]. In order to describe structure, attribute grammars adopt context-free grammars as their basic skeletons. Context-free grammars use syntax rules to define the syntax of the language. Each rule explains the structure of a composite construction (i.e., a non-terminal symbol) in terms of simpler composite constructions, as well as in terms of primitive constructions (i.e., terminal or non-terminal symbols). Using syntax rules it is possible to impose a tree shaped structure on each sentence in the language. This structure is the sentence's parse tree, which represents how to apply the syntax rules to generate such a sentence. To add processing capabilities, attribute grammars associate semantic attributes with the syntax symbols and semantic equations with the syntax rules. There are two kinds of attributes: synthesized and inherited. The value of a synthesized attribute represents the meaning of the symbol. The value of an inherited attribute provides additional contextual information. Semantic equations, in turn, indicate how to compute the values of semantic attributes in syntax rules using the values of other semantic attributes.

Attribute grammars enable semantic evaluation on attributed parse trees (i.e., parse trees along with the semantic attributes for each node). Semantic evaluation consists of assigning values to the attributes of each node, and it is driven by the dependencies between attributes. Therefore, the only constraint imposed by the attribute grammar on semantic evaluation is that, before computing the value of an attribute, the values of the attributes on which it depends should already have been computed. Aside from this basic constraint, evaluation order does not matter. Starting from the basic formalism, as explained in [120], the community of computer languages has proposed a wide range of tools to implement language processors from a specification in the form of attribute grammars. These tools range from classic systems like GAG [82], FNC-2 [80], ELI [63] or Elegant [16] to more modern systems such as LISA [70] [108] [118] Silver [181] or JastAdd [101]. As already indicated, these tools take, as input, a specification based on the formalism of attribute grammars and generate, as output, a language processor for the language defined. Also, many of these tools support different metalanguages extensions to enrich the basic formalism of attribute grammars (e.g., modules [84], generics [148], higher order [172], object [68] or aspects [128] [129]), which enable the creation and maintenance of complex specifications.

The code generated by these tools is usually highly optimized, following a static approach in their evaluation and storage strategies, which are determined as a result of a static analysis of the input specification [1]. In short, the code generated is not likely to be inspected or modified by people.

7.2.4.2 Tools based on translation schemata

Besides attribute grammars, translation schemata are also widely used as specification formalism in a wide variety of tools for generating language processors.

Translation schemata, in turn, are another extension of context-free grammar that augments syntactic rules with semantic actions: chunks of code in a host programming language to be executed during parsing [5]. These actions can consult and update semantic values that augment the parsing state. There are many parsing models, but the following two are especially relevant:

- Bottom-up translation schemata. These schemata assign a semantic action to each syntactic rule, which will be executed when the rule is used in a reduction during a bottom-up recognition of the input sentence [5].
- Top-down translation schemata. These schemata interleave semantic actions in the rule's RHS, which are executed as they are discovered during the top-down recognition of the input sentence [5].

Language processors construction tools based on translation schemata normally adhere to a specific model of translation scheme. In particular:

- Tools for generating bottom-up, shift-reduce, translators use LR bottom-up translation schemata as input. Therefore, the resulting processors use a stack to store the semantic values. They can also use global variables to store additional semantic information. This type of tools is normally restricted to translation schemata formulated on LR, usually LALR(1), context-free grammars [5], although there are such tools that accept more general types of grammars [29] [106]).
- Top-down translation schemata are used as specification formalism for generation tools of, top-down, predictive descent translators. Since many of these tools generate recursive descent parsers, semantic information is managed as input and output parameters of the subprograms generated, as well as local variables. As in the bottom-up ones, processors generated not explicitly construct the syntax tree. These tools impose strong restrictions on grammars: LL grammars. Although modern generation tools such as ANTLR, provide extensions to LL(k) grammars (in particular, support to the LL(*) parsing method, which provide unlimited prediction [125] [124]), they are not capable of handling left recursion properly, for example. Yet these tools provide a more natural way to manage the inherited information.

Examples of tools that generate top-down translators are JavaCC [91], ANTLR [124] and COCO/R [114]. The tools that generate bottom-up translators are based on bottom-up translation schemata. YACC [154], Bison [97], CUP [14], Tatoo [33], SableCC [52], Beaver Copper [173] or YaJco, are examples of this type of translator generators.

7.2.4.3 Implementation of specifications based on attribute grammars with translation schemata

This thesis proposes the use of conventional translators generation tools (based on translation schemata) to carry out the implementation of specifications based on attribute grammars, in order to allow the improvement of specification-driven development processes by rapid prototyping and systematic implementation of the specifications. For this purpose, there are different techniques to use translators generation tools based on translation schemata to implement language processors from their specifications based on attribute grammars. Usually, these techniques are based on the coupling of semantic evaluation and parsing, aspect which has been addressed extensively as a way of implementing different kinds of attribute grammars (see [7] to an introductory tutorial). The techniques themselves can be classified into two groups, according to the type of generator used: techniques oriented to top-down translators, and techniques oriented to bottom-up translators. Below each one is revised.

7.2.4.3.1 Techniques oriented to top-down translators

The works [5] [7] show how to use generators to implement top-down translators based on LL-attributed grammars: L-attributed grammars built on context-free LL grammars. The technique involves calculating inherited attributes of each symbol on the right side of a rule immediately before of such a symbol, as well as calculating synthesized attributes of the symbol in the left side at the end of the production. The implementation of more general grammars involves the transformation of the original specification, using mechanisms and transformations able to transform an attribute grammar into an equivalent LL-attributed grammar. Once transformed, the developer is able to use the transformed specification as input for the top-down translator generation tools to obtain the specified processor.

Some of the transformations oriented to obtain the equivalent LL-attributed grammars from the initial specification are elimination of left recursion and factorization [5], as well as backpatching [3] for dealing with non-l-attributed specifications.

7.2.4.3.2 Techniques oriented to bottom-up translators

As with the tools to generate top-down translators, it is possible to construct language processors from attribute grammars using bottom-up translator generation tools. Although the general models for coupling attribute evaluation with parsing allow executing evaluation operations both during reduction and shift actions [7] [8] [107], which makes it possible to support the class of LR attributed grammars, most of the bottom-up translation generation tools only allow semantic actions associated with the reduction of syntax rules. As a consequence, they only enable the direct implementation of s-attributed grammars with underlying LALR context-free grammars. Indeed, the values of the synthesized attributes of the left symbols are calculated in those actions associated with the reduction of the rules. Also, in certain situations it is also possible to implement some grammars with

inherited attributes using *markers* [5], new non-terminal symbols defined by right-empty productions, and use them to store inherited attributes. To do this so, the context-free grammar has to be modified by placing markers at suitable places of production, which, in turn, can result in the loss of LALR grammar character. This fact makes this technique a difficult one to implement and systematize.

7.2.4.4 Educational uses of development tools for language processors

The previous sections show how the professional tools for the development of language processors are useful in the practical development of translators. However, in education these tools may not necessarily have the right desirable characteristics for students. Still, there are different educational strategies that use these tools to present different concepts and techniques to students. In fact, in section 7.2.2.2.2, it has already been shown how the tools for building language processors can play an important role in the teaching-learning of the subject. However, in that section the tools were used as auxiliary tools to produce processors that can be instrumented through simulation techniques. Nevertheless these tools can also play a central role in the teaching-learning process, as it is discussed in the following subsections.

7.2.4.4.1 Educational uses of generic tools

As already indicated, the generic language processor development tools are widely used for the professional development of language processors. Moreover, within certain parameters and conditions, as was also indicated, these tools can be used for educational purposes within the subjects of Language Processors. For example:

- LISA is a tool based on attribute grammars that has been used for educational purposes [109]. This use is motivated by different visualizations and simulators provided by the tool, it has a semantic evaluation process debugger, a simulator of the construction of syntax tree, a visualization of the dependency graph, etc.
- ANTLRWorks [26] is a development environment for ANTLR that provides a grammar editor and an interpreter to help the user during the prototyping and a debugger to identify errors in the specification. Such a debugger is a very powerful tool, making it possible to show the construction of decorated syntax trees. On the other hand, the debugger is able to identify problems of ambiguity in grammars.

7.2.4.4.2 Specific tools for education

These tools are tools for generating language processors, but this time explicitly dedicated to their use in education. As examples of such environments we have contemplated:

- VCOCO [130], a visualization software for COCO, a language for the creation of language processor, which generates language processors from specifications based on EBNF grammars annotated with semantic actions.

The simulator shows to the student the grammar, the sentence, the generated machine code and the source code of the program being debugged. The tool highlights simultaneously the different parts involved in each step of simulation to let students understand how all the parts of the processor work as a unit during the simulation.

- PAG (Prototyping With Attribute Grammars) [158] is a development environment that is capable of processing specifications based on attribute grammars to generate prototypes of the language processor specified. The specifications are written using a syntax embedded in Prolog [160], very similar to the attribute grammar notation. The tool then generates the processor prototype, and use it to analyze sentences allowing students to inspect all possible decorated syntax trees, navigate along the nodes and check the semantic values of the attribute instances associated with those nodes.

7.2.4.5 Discussion

This section has been able to confirm the existence of a wide range of tools for generation of language processors based on attribute grammars and translation schemata. For this purpose:

- The language processors generation tools based on attribute grammars provide an appropriate framework for the development of language processors led by declarative specifications. However, implementation generated does not accurately correspond to the starting specification, due to various optimizations that are performed during generation.
- Moreover, the tools for the generation of processors based on translation schemata supported lower level formalisms, more oriented to describe the implementation of the processors by domain-specific formalisms (linked to grammars) than to declaratively specify these processors.

In addition, we have analyzed different techniques and strategies for implementing language processors based on attribute grammars using the potential of generators based on translation schemata. These techniques and implementation strategies have a clear interest from an educational point of view, because they allow students to understand how to systematically transform the specification of a language processor into its final implementation. However, the analyzed approaches presented share a main limitation: they only are able to handle particular types of attribute grammars (LL-attributed grammars in the case of top-down translator generators, s-attributed grammars defined on LALR context-free grammars in the case of bottom-up translators). While it is possible to deal with more general grammars, it will usually be necessary to apply specific techniques for each particular specification, which are usually error-prone and which thus can hinder the final student learning.

It has also shown the educational value of processors generation tools in typical Language Processors courses with positive results for students. These results were obtained thanks to different language processor generators that provide different visualizations and enriched debuggers for helping students to understand the processing more intuitively. The educational use of LISA and ANTLRWorks are two

examples of this strategy. On the other hand, there are generators oriented to education, as PAG and VCOCO, which are specifically designed to help students understand how their own language processors work using simulators. However, from our point of view, these tools have the following deficiencies:

- Tools based on translation schemata (e.g., ANTLRWorks and VCOCO), focus particularly on issues related to the implementation of the processors, putting aside aspects of the specification using higher-level formalisms.
- Tools based on the formalism of attribute grammars (e.g., LISA or PAG) do not allow students to interactively engage in the process of semantic evaluation, which, in our view, is an essential aspect to enhance the understanding of the fundamentals of language processor specification. Also, they do not offer mechanisms that allow students to understand how to convert specifications into effective implementations of processors, using for this purpose, systematic implementation strategies. On the contrary, translators generated are based on implementation in which the connection between the specification and implementation is lost (e.g., highly optimized implementations of semantic evaluation processes [69] in the case of tools based on attribute grammars as LISA, or a generic attribute grammar interpreter written in Prolog in the case of PAG [158]).

7.2.5 In conclusion

Throughout this section, we have analyzed different aspects related to teaching and learning of concepts of the design and implementation of computer languages: strategies for the teaching-learning of Language Processors subjects, use of educational simulations and serious games, and specific tools to build processors.

Teaching strategies of language processors analyzed follow different approaches to teach to students the concepts and techniques necessary for the design and development of a language processor. The most widely used strategy takes as its cornerstone the development, by students, of a processor for a programming language. This objective is the main thread of the lectures and laboratory sessions. Meanwhile, other strategies seek to motivate students proposing attractive projects based on other languages, or concepts presented radically different from the traditional way by experimenting with actual language processors. While all educational strategies are relevant to teach the construction of language processors, within the framework of this thesis they have a common flaw: these strategies focus primarily on the implementation stage of the processors at the expense of the prior specification state, essential to articulate properly the development phase.

Regarding the interactive simulations, in this section we have analyzed how different software tools exploit the educational benefits that provide visualizations and interaction to enhance the educational experience of students, helping them understand the different processes and artifacts that are part of their educational domain. We also have analyzed different tools related to the field of Language Processors. Most of the analyzed tools focus on issues related to the implementation of these processors, but we have observed a significant lack of simulators focused on aspects related to the specification.

Regarding serious games, we have analyzed how these approaches take advantage of the great attractive of this type of artifacts to motivate students to learn different concepts in an entertaining way. Though the range of this type of software in computer education is broad, it is not true for the construction of language processors. This is due mainly to the difficulty of finding a suitable game mechanics for the concepts that compound the course that can be, in turn, entertaining and attractive for students. As an exception, we have analyzed the JV²M system, which yet again emphasizes aspects of implementation, rather than specification ones.

Finally, we have analyzed different tools for generating language processors from specifications based on attribute grammars and translation schemata. These tools are suitable for experienced developers to design and implement language processors, enabling them to develop complex translators in fast and effective ways. In this section, we have also revealed the educational potential of these tools, especially those that provide students with interactive visualizations of different data structures involved in language processing (parse tree, parse stack, etc.). However, the tools based on translation schemata are more geared to the implementation of the processors than to its specification. Meanwhile, although the tools based on attribute grammars operate over higher-level specifications, it was observed that, on the one hand, they do not allow students to interactively involve in the process of semantic assessment and, secondly, they do not generate implementations that preserve the specification. This second aspect, the generation of implementations that preserve the specification, can be addressed by implementing attribute grammars on translation schemata, although, as also discussed in this section, the systematic application of these techniques, in its current form, is restricted to particular types of attribute grammars, while handling more general types of grammars require of nontrivial and error-prone transformations and techniques.

In conclusion, the different approaches, strategies and software tools analyzed in the field of construction of language processors show that, from an educational point of view, there is a significant lack of attention paid to teaching concepts related to the specification of language processors, opposite to the attention paid to implementation techniques. This thesis aims to provide a partial solution to this gap by: (i) the formulation, and computer support, of a strategy to facilitate the understanding of the underlying fundamentals of specifications mechanisms based on attribute grammars in the early stages of teaching and learning process, (ii) the systematic implementation of these specifications using tools based on translation schemata, and (iii) integration of the previous two aspects in an educational-oriented development tool of language processors driven by specifications and based on debugging.

7.3 Objectives and work plan

As already mentioned in previous sections, the proper development of a language processor depends heavily on: (i) an initial specification phase correctly performed, where the different aspects of a language processor have to be rigorously and unambiguously specified by the formalism targeted to each of these

aspects, and (ii) the systematic application of implementation techniques driven by the specifications, which lets transform these formal descriptions into the final working implementation of the processor. However, the students who start in the subjects of design and implementation of computer languages lack the skills necessary to produce adequate specifications and, therefore, address the development of its processors based on incomplete or incorrect designs. This fact is especially critical as regards key aspects in the development of a processor, such as syntax-directed translation specification processes using, for example, an attribute grammar-based formalism. Furthermore, during the implementation phase, students often obviate specifications addressing development arbitrarily, based on their experience as programmers, rather than in the systematic application of implementation techniques driven by these specifications.

Thus, it is necessary that students in subjects related to the development of language processors properly assimilate the importance of the specifications in the development process, being able to understand, produce and implement them effectively during the construction of the processors. However, as has been evidenced in the previous section, there is a clear gap about this in relation to the different teaching strategies concerning development of language processors, as well as the different software tools for teaching these concepts (e.g., simulations and serious games), as these focus mainly on aspects of implementation and internal details of algorithms of the processors, in detriment of the specification aspects mentioned above. Therefore, this thesis proposes to address that gap on tools and strategies that affect the understanding, production and exploitation of the specifications during the teaching-learning process on those matters.

Thus, in this section, we present the objectives to be achieved in the proposed thesis (subsection 7.3.1), and we describe the work plan to be followed to achieve these objectives (subsection 7.3.2).

7.3.1 Objectives of the thesis

As mentioned above, often students in introductory language processors courses do not consider important the aspects and concepts related to the specification and the systematic development based on the specification. This fact is evidenced, for example, in the development of language processors posed as final projects of the course, where it is common to observe how the task of specifying and developing these processors is tedious and unsatisfactory for students, due to the large number of problems encountered during development. So students develop their processors using *ad hoc* methods rather than systematic approaches focused on the specifications. Given these considerations, the work proposed in this thesis is focused on providing educational strategies and software tools to enhance the students understanding of the language processors specification formalism, and on assisting students in applying systematic language processors development techniques from their specifications. In particular, given the importance of the specification in the development of processors based on syntax-directed translation, this thesis will focus on the formalism of attribute grammars. To do this so, this thesis presents the following three objectives:

- The first objective focuses on developing an educational strategy to facilitate students to understand basic concepts in the specification of a language processor using the formalism of attribute grammars. This strategy will be focused on early stages of learning, and will be supported by a set of software tools that support the new trends in eLearning to present these concepts (in particular, in serious games and interactive simulations). Moreover, the achievement of this goal will require different assessments of the pedagogical adequacy of the teaching-learning strategy and the associated software tools.
- The second objective is focused on providing a development strategy of language processors, based on a specification in the form of a general noncircular attribute grammar to produce functional prototypes of the language processor specified. These prototypes will be supported by conventional tools for the generation of processors, based on translation schemata. Additionally, the educational usefulness of this approach will be evaluated.
- Finally, the third objective focuses on providing a software platform for the development of language processors from its specification through the formalism of attribute grammars integrating the main results obtained in the preceding two objectives: on one hand, the understanding strategy of attribute grammars formalism, and, the coding of attribute grammars specifications into language processor using conventional development tools. Using the comprehension strategy, this platform will provide advanced tools for debugging language processors specified by attribute grammars. Moreover, the tool will use the coding patterns for general noncircular attribute grammars into translation schemata to generate the aforementioned processors. In addition, this third objective will also focus on assessing the educational value of the platform developed.

The following sections go inside each of these objectives, laying the foundation for the thesis work plan, as outlined in the following section.

7.3.1.1 Strategy for the comprehension of attribute grammar-based specifications

Specification formalisms of language processors are traditionally one of the corner stone used in a course of language processors. Students need to handle them fluently, because those will be used throughout the course to specify and design the different components that compound this processor (scanning and parsing, construction of the symbol table, static semantics analysis, code generation, etc.). This aspect of the thesis focuses specifically on the formalism of attribute grammars discussed in the previous section. Usually, this formalism allows specifying the syntax of a language and its semantics. Besides, it presents a distinctive computation process, driven by the dependencies between attributes, which determine how the associated semantic values are calculated. This aspect of attribute grammars is often particularly difficult to assimilate for students.

As we saw in the previous section, there are different strategies and software tools to improve the teaching and learning of Language Processors subjects. However:

- Although the different strategies used in the previous section have shown their educational value in the field of education of the subject, such strategies lack of an essential aspect from the point of view of the main focus of this thesis: those focus on issues related to the implementation of the processors, relegating to a secondary role the specification part of the mentioned processors, regardless the importance played by these specifications in the subsequent development process.
- The same fact is true for the software tools analyzed, both interactive simulators and serious games related to teaching Language Processors. Both are strongly oriented to aspects directly related to the implementation of language processors.

Therefore, given the importance of the basic concepts of specification based on the formalism of attribute grammars, even from the early teaching-learning process stages, and the difficulties the students may encounter in their learning, and in their application in the design and development of a language processor, the first objective of this thesis is formulated as:

> *Objective 1. To develop an educational strategy to facilitate understanding of the formalism of attribute grammars in the early stages of learning, support this strategy through educational software tools based on serious games and interactive simulations, and validate the educational adequacy of both the strategy and software tools through assessments with teachers and students of a Language Processors course.*

7.3.1.2 Strategy for the implementation of attribute grammars as translation schemata

Another of the most problematic aspects that students in a Language Processors course face is obtaining the implementation of a processor from its specification. This process may involve non-trivial transformations in the specification, where students could make multiple mistakes that result in a final incorrect implementation, detached from the original specification.

Specifically, as noted in the previous section, in the case of specifications based on attribute grammars, although there are tools for generating language processors directly based on this formalism, these tools carry out automatically the generation process from the input attribute grammars, resulting, in this way, a completely opaque process for students, and there are no easily identifiable direct relationship between the generated implementations and the specifications provided.

Therefore, from an educational point of view, it is important that students learn how to relate their attribute grammar-based specifications with the most significant translation models. For this, and as discussed in the previous section, the most common approach is to apply sequentially transformations to the original attribute grammar in order to obtain an equivalent translation schememata that can be implemented directly using more conventional processors generation tools,

as CUP or JavaCC. However, and as also shown in the previous section, these transformations can be systematized only for certain types of attribute grammars, requiring, in other cases, *ad hoc* treatments heavily dependent on the particular specification solutions. The result is an error-prone process, which can distort the final learning objective: to enable students to relate specifications based on attribute grammars with conventional translation models. Once available the implementations of attribute grammars as translation schemata, students can apply transformations aimed to improve the efficiency of these. However, the critical aspect of this process is to find the aforementioned coding patterns of arbitrary non-circular attribute grammars as translation schemata. Therefore, the second objective of the thesis is formulated as follows:

> *Objective 2. To develop a strategy that allows students to obtain functional prototypes of language processors through direct coding of non-circular arbitrary attribute grammars as translation schemata, to support the strategy through an appropriate software library, and to evaluate the education usefulness of the formulated strategy and the library that supports it.*

7.3.1.3 Educational language processor development environment based on attribute grammars

Proper assimilation of the language processor development process driven by specifications based on attribute grammars requires, as noted in previous sections, the combination of two basic aspects: (i) a deep understanding of the concepts and mechanisms underlying attribute grammars as a specification formalism, and (ii) being able to relate the specifications to their implementations with conventional translation models. It is, therefore, important that students are able to handle both aspects fluently, but it is even more important that students understand how both aspects are combined to specify and develop a processor. However, as discussed in the previous section, the tools for building language processors not adequately cover these aspects. Indeed:

- Tools based on translation schemata address the description of the translators, instead of a description of their specifications by more declarative formalisms.
- In its turn, although the tools based on attribute grammars can agree to such declarative specifications, those do not include debugging mechanisms aimed to students. Also, they generate highly optimized implementations that do not preserve the specification.

Therefore, from an educational point of view, it is desirable to provide a tool for the development of language processors from their specifications in the form of attribute grammars that combines the advantages of similar tools, in what refers to directly accept this sort of attribute grammar-based specifications, with the advantages concerning aspects of understanding and implementation of these specifications addressed in the objectives outlined in the previous sections. Thus, the third objective of the thesis is formulated as follows:

> *Objective 3. Design, implementation and evaluation of a language processor development environment based on specifications as attribute*

grammars, which will integrate a debugging model inspired by the comprehension strategy developed for the objective 1, and which will be able to generate implementations of processors by applying the coding patterns of arbitrary non-circular attribute grammar as translation schemata developed for the objective 2.

7.3.2 Work plan

Given the above considerations and the objectives proposed in the previous section, we can say that the overall purpose of this thesis is to provide new approaches and tools to enhance the teaching-learning process of language processors specification and its subsequent implementation from those specifications. These approaches and tools exploit the discovered advances in education and *eLearning* to present these concepts in an attractive and educationally effective way.

From the objectives outlined in subsection 7.3.1 it is possible to derive a series of specific activities, which address both methodological and technical aspects that structure the core of this thesis work, and crystallize finally in the creation of various educational strategies and software tools that support them. The following subsections describe such activities.

7.3.2.1 Activities concerning the strategy for the comprehension of attribute grammar-based specifications

To achieve the objective 1 it is necessary, first, to formulate an educational strategy for teaching the fundamentals of specification of language processors using the formalism of attribute grammars. After, we will develop an educational software system that supports the educational strategy formulated using the latest trends in eLearning. Subsequently, the educational strategy and the educational software system will be validated, through its implantation in a course. Finally, the process model followed in the formulation of the educational strategy and the associated software system will be abstracted to make it possible to support other specification formalism and other different educational fields.

7.3.2.1.1 Formulation of the educational strategy

One of the main aspects to be covered in the first objective is to develop an educational strategy for teaching basic concepts of attribute grammar formalism related with the specification of language processors in the early stages of learning. Indeed, the analysis of different strategies oriented to teaching Language Processors in the preceding section has shown how there is no similar strategy specifically geared to the formalism of attribute grammars, with the exception of strategies implicit to tools such as LISA or PAG, tools which, moreover, already presuppose prior knowledge and skills in handling such formalism. Thus, the formulated strategy will focus specifically on the issues related to understanding the fundamentals of attribute grammars, and, in particular, their characteristic computing process.

The objective of the strategy will be to ensure that students internalize the basic mechanisms of semantic evaluation derived from the formalism, and, in particular, the computing style derived from the dependencies between attributes, and not the specific processing performed from carrying out the parsing of sentences. To do this so, the strategy will advocate the active immersion of the student in the process of semantic evaluation, through the proposal of specific exercises related to the evaluation, in whose resolution the student must find possible ways to determine the values of the attributes in key examples carefully selected by the teacher.

7.3.2.1.2 Development of the educational supporting software

At this point, with educational strategy set, we will address the development of educational software system that supports this educational strategy to improve the teaching-learning formalism of attribute grammars in the early stages of this teaching-learning. So, considering that the software will be used by teachers and students, the software system will provide the tools to support every aspect of the formulated strategy, as well as for supporting each of the roles involved in the strategy (teachers and students). In particular:

- The system should allow teachers to propose to students significant cases designed to improve the comprehension of the basic concepts underlying attribute grammars.
- The system should provide, also, the conversion of those cases into suitable artifacts to assist students in the comprehension process of these key concepts. In particular, we will search for mechanisms that automate as much as possible the production of such artifacts.
- Finally, the system should provide tools to the teachers to analyze the behavior of students while using the aforementioned artifacts.

The artifacts themselves should actively involve students in the semantic evaluation processes underlying attribute grammars, in a way that should be consistent with the proposed educational strategy. To do this so, approaches based on serious games and interactive simulations are considered, in order to promote the active participation of students in these semantic evaluation process, powering, thus, the understanding of these key aspects of the formalism.

7.3.2.1.3 Validation of the educational strategy and the associated educational software

Once formulated the educational strategy and after completing the development of the educational software system that supports the teaching-learning strategy, it is necessary to validate the educational method to verify its educational usefulness with the students and teachers of matters related to the development of language processors. To this end, we shall apply the following procedures:

- To pose different assessments with teachers, both specialists in language processors, as teachers focused on other areas of Computer Science related with Artificial Intelligence and Software Engineering. These assessments will cover different aspects such as: appropriateness of the educational strategy, perceived usefulness of the software system developed, usability

of software tools for content creation and evaluation of the activity performed by students, etc.

- To perform different experiences involving students of a language processors course. In these studies we will analyze the perceived usefulness of the educational strategy and the educational software system, its educational effectiveness, etc. We will carry out these studies at short-term, with specific experiences during the course, and at long-term, for several academic years, to assess the impact of the strategy throughout the course.

Taking into account that teachers and students will be actively involved in both the educational strategy, and the software that accompanies it, it is important to consider the assessments and results provided by both during the experiences and the studies to be undertaken. The results obtained from these studies will provide enough information to validate the software system developed and the designed educational strategy, and to improve this system and the underlying strategy.

7.3.2.1.4 Formulation of the process model

In order to allow the systematic addressing of other approaches and specification formalism, finally a process model as an abstraction of the steps followed in the developing of the educational system raises. This process model will start from batteries of exercises proposed by teachers to enhance teaching and learning, and will be oriented to build, maintain, and evolve an educational system that adopts the following biases:

- We will apply a generative approach that will allow teachers create educational software used by students from batteries of exercises.
- It will be necessary to provide teachers with software tools for editing the exercise batteries to be used to generate the software products and tools, and to monitor and assess the performance of students through the use of the software products generated.
- We will consider different software assessment methods that enable developers and teachers to guide the development of educational software system to fit the needs of end users.

The model itself will emerge as an abstraction of the process followed in developing the above mentioned system, the analysis of the different aspects that compound the development method followed, and abstracting these aspects to formulate the model of development, so that it can be applied to other contexts related to the subject of Language Processors (and ideally to other matters in which it makes sense to empower comprehension strategies of basic concepts and procedures in the early stages of teaching-learning).

7.3.2.2 Activities concerning the strategy for the implementation of attribute grammars as translation schemata

To achieve the objective 2 is necessary, first, to formulate a set of patterns to encode non-circular arbitrary attribute grammars on conventional processor development tools based on translation schemata. Once the coding strategy is set, we will develop a software library to support this strategy that will facilitate the

encoding of grammars using conventional development tools. Finally, the formulated strategy and the software developed will be evaluated using its implantation in a Language Processor course in order to validate the educational adequacy of both.

7.3.2.2.1 Formulation of the development strategy

The main task of objective 2 is to devise a development strategy for direct coding of language processor specifications using the attribute grammar-based formalism into conventional language processor building tools. By applying this strategy, students will be able to get functional prototypes based on translation schemata. Indeed, in the previous section, we have analyzed different approaches to the implementation of attribute grammars as translation schemata. However, as discussed above, such approaches suffer from various shortcomings hindering educational utility (they are only systematized for particular types of grammars, required laborious and error-prone transformations, and specific *ad hoc* and particular solutions in the case of more general types of grammars). Thus, during this task, we will put special emphasis in getting a coding strategy that, on the one hand, would be able to systematize the encoding of arbitrary non-circular attribute grammars, and, moreover, would feel natural and intuitive for students.

The main objective of the proposed strategy is that students understand the relationship between the language processor specified and the implementation obtained. To do this so, the strategy will involve students directly in the encoding process, so they will be aware of the existing relations between the original attribute grammar and the implementation resulting from the application of the proposed coding approach.

7.3.2.2.2 Development of the supporting software

The attribute grammar coding strategy as translation schemata raised must rely on language processors generation tools based on translation schemata, so it will be necessary to develop a software library to facilitate the integration of the semantic evaluation determined by attribute grammars into the codified translation schemata. This library will facilitate the implementation of the coding patterns, and will allow students to be able to identify, both syntactically and semantically, the relationship between the implementation obtained as a translation scheme and the original attribute grammar.

7.3.2.2.3 Validation of the strategy and the supporting software

Once formulated the strategy and implemented the supporting software, it will be necessary to validate the educational, effective and perceived usefulness by teachers and students. For this purpose, a similar procedure to the one performed in the first objective will be followed:

- It will be necessary to consider experiences with teachers specialized in teaching the subject of Language Processors aimed to evaluate the perceived educational utility for teachers as well as the appropriateness of the strategy in a typical course of this subject.

- On the other hand, experiences with students will be raised to observe the educational effectiveness of the development method proposed compared to other development approaches. Also, during these experiences, the student opinions on the proposed development approach will be obtained, as well as the educational utility perceived by them.

The results obtained from teachers and students will provide enough information to verify the educational effectiveness of the approach, as well as the perceived utility and assessment by the main actors involved in the approach. In this way, we can check the validity of the approach and its suitability for its use in Language Processors courses, as well as propose possible improvements to this approach.

7.3.2.3 Activities concerning the creation of the educational language processor development environment based on attribute grammars

To achieve the objective 3 will be necessary, first, to build a software platform for the development of language processors specified by the formalism of attribute grammars that: (i) allow students to understand how their own specifications based on this formalism work, relying on the strategy set out in the objective 1, and (ii) to generate implementations of the specified processors based on the development strategy raised in objective 2. It will also be necessary to validate the software system developed and its application on a Language Processors course. To do this so, different experiences between teachers and students in these subjects will be organized.

7.3.2.3.1 Development of a software platform for the creation of language processors

To achieve objective 3 raised in the previous section, it is necessary to design and develop a language processor development environment based on the specification formalism of attribute grammars, which combines the two approaches outlined in objectives 1 and 2: to facilitate the understanding of the specifications and the subsequent implementation as translation schemata. For that:

- The platform will provide tools to facilitate the understanding of basic concepts of the attribute grammars specified by the student through a powerful visual debugger based on the educational strategy formulated in Objective 1.
- The platform will generate, in turn, language processors based on the coding mechanisms of the objective 2. This coding strategy is carried out automatically by the tool, allowing students explore the translation scheme codified and find the relationship with the original attribute grammar-based specification.

It is important to note, finally, that the platform, unlike other platforms oriented to experienced developers (eg LISA), will be doted of an essential educational flavor, and will be completely connected to the strategies developed in the first two objectives of this thesis.

7.3.2.3.2 System validation

When the development of the platform will be completed, it will be necessary to validate the software created. As in previous cases, different experiences between teachers and students of the courses in Language Processors will be organized to collect data on the perceived educational utility of the tool. With these data, we will be able of validating the usefulness of the platform as well as of obtaining useful information for enhancement.

7.3.3 Conclusions

In order to encourage improvements in the acquisition of the essential concepts and techniques for the specification and systematic implementation of language processors driven by their specifications, in this thesis we raised objectives oriented to:

- The formulation of a strategy to facilitate the understanding of basic concepts of the attribute grammars formalism, supported by a software system that makes use of new trends in eLearning and education.
- The formulation of a development method for coding non-circular attribute grammars as translation schemata, which aims to facilitate the application of systematic development techniques of language processors from their specifications.
- Creating a development platform that combines the implicit approaches in the two previous objectives for students to be able to understand their own specifications given in the form of attribute grammar and their subsequent implementations in the form of translation schemata.

Thus, in the next section, we will present a discussion of the content of the articles that accompany this thesis in order to integrate their content and relate them to the objectives presented in this section. On the other hand, in subsection 7.5 we will analyze to which extent the objectives have been fulfilled with respect to the results presented in articles and also we will describe the main lines of future work.

7.4 Discussion of the contributions of the articles

This section contextualizes the research results obtained during the development of this thesis based on the publications that support it. Subsection 7.4.1 contextualizes the results concerning the educational strategy and the related software to improve understanding of basic concepts of attribute grammars (objective 1 of the thesis). Subsection 7.4.2 contextualizes the results for the encoding of arbitrary non-circular attribute grammars into translation schemata (objective 2). Subsection 7.4.3 summarizes, finally, the results for building language processor development tools based on attribute grammars integrating these strategies for the understanding and encoding of attribute grammars (objective 3).

7.4.1 Strategy for the comprehension of attribute grammar-based specifications

The educational strategy designed to assist teachers and students in the teaching-learning of basic concepts of attribute grammars formalism is supported by a software system that allows teachers to provide batteries of exercises based on language processing problems where students must: analyze and understand the proposed language, perform the processing of the sentence provided, and demonstrate that they are able to emulate the computational model of the formalism properly. In order to motivate the students, it was decided to present these exercise batteries with interactive simulators taking advantage of the new trends in eLearning.

Subsection 7.4.1.1 addresses the educational strategy proposed to facilitate the acquisition of basic concepts of attribute grammars. Subsection 7.4.1.2 addresses the software system developed to support this strategy. Subsection 7.4.1.3 addresses the different experiences conducted among teachers and students to validate the educational strategy and software system that supports it. Finally, subsection 7.4.1.4 addresses the process model to guide the development of educational software systems based on generative approaches similar to that proposed in subsection 7.4.1.2.

7.4.1.1 Formulation of the educational strategy

The educational strategy proposed to facilitate the learning of concepts related to the formalism of attribute grammars focuses mainly on semantic evaluation aspects of this formalism. As described in [135] [138] [137] [139] [141] [142], the educational strategy calls for the provision of batteries of exercises in which language processing tasks are proposed. These tasks will be described by a language specified by an attribute grammar, a sentence for that language and the decorated syntax tree derived from the analysis of the sentence. The students objective is to provide an evaluation order of each of the attribute instances that decorate the syntax tree and its value. In this way, students are able to understand how the semantic evaluation is performed in accordance with the formalism.

7.4.1.2 Development of the educational supporting software

In order to support the educational strategy proposed in the previous subsection, the development of a software system has been carried out for: (i) building exercise batteries of the kind described above, (ii) the generation of interactive simulators based on those exercises and (iii) assessment of student progress. In [135] [138] [137] [141] [142] we described the different software tools that make this educational system and the evolution of these tools. This toolkit, which is a direct result of the requirements set out in subsection 7.3.2.1 to achieve the objective 1 of the thesis, consists of an *authoring tool*, *generators simulators* and an *analysis tool*.

The *Authoring tool* provides enough expressiveness to describe the types of exercises that students will solve, using the generated simulators, which are described in subsection 7.3.2.1.1. Initially, and as described in [135] [138] [137]

[142], we developed a tool that allowed edition of the parse tree for the input sentence and the dependency graph among attributes. Next, and as described in [135] [138] [141], we improved the tool to obtain a more usable tool that provides educators with an attribute grammars editor where they only need to provide the attribute grammar specification and a sentence. With this information, the tool automatically generates all the possible solutions (the syntax tree and the dependency graph, required to check the semantic evaluation orders provided by students).

Simulators generators are able to generate different simulators from exercise batteries created by the authoring tool. During the development of this thesis we have built two different generators:

- A generator that automatically generates serious games from exercise batteries. The serious games generated are based on a video game puzzle type. The game presents to the user a labyrinth, representing the parse tree, where it is possible to find rooms that represent tree nodes and corridors that connect them, related to the arcs of the tree. Within these rooms there are tables with boxes representing the attributes of each node. These boxes may contain objects, which correspond to the values of those attributes. The user objective is to move these objects to other boxes, guided by the dependencies between attributes specified in the grammars of the exercises, in order to set the value of each of the labyrinth boxes. Thus, the student provides, implicitly, the evaluation order of the attributes of the tree. The generator and the generated game type are described in [135] [138] [137] [141] [142].
- A second generator that generates interactive simulators. Simulators generated present all the information contained in each battery in an attractive way to students. The language described by the attribute grammar, and the sentence to process are presented textually to students. The syntax tree is represented graphically forming the central axis of the simulator. The attributes of the nodes are represented by squares, which appear when a node in the syntax tree is selected. Finally, students can view information about selected attributes. The mechanics of the simulator is identical to serious games, while avoiding the playful component, which can distract students from the ultimate goal of emulating the semantic evaluation process. Thus, students should explore the nodes of the syntax tree for those attributes whose value has not been calculated and, guided by the dependencies described in the attribute grammar, move the necessary values to those attributes that need to be calculated. In addition, for each of the calculation performed by the student, the tool provides information on the validity or invalidity of these, to guide students in solving the exercise. This type of generator and the generated simulators are described in [135] [138] [139].

Finally, the *analysis tool* provides teachers with an environment to study and evaluate the actions taken by students during the resolution of the exercises batteries with the generated simulators. During the thesis, and as described in [135] [138], a first simple analysis tool was built, which textually shows the actions performed by the students. Then the tool evolved into a one that visually presents

the representation of the syntax tree, and the actions taken by the student during the process of solving each problem. This tool is described in [135] [138] [141].

The tools described above compound the educational system *Evaluators*, which is capable of generating both educational games and interactive simulators based on visualization from exercise batteries oriented to teaching basic concepts of attribute grammars. As described in [135] [138], the generators operate on the same kind of battery of exercises. Therefore, teachers can generate serious games or interactive simulations from a single source of exercises. Likewise, the analysis tool also works interchangeably on the outputs provided by the two types of simulators (serious games and interactive simulations). The system itself gives software support to the educational strategy outlined in the previous section providing tools for creating exercises, tools for creating simulators and tools for evaluating the solutions given by students.

7.4.1.3 Validation of the educational strategy and the associated educational software

In order to validate both the educational strategy, and the educational software that supports it, during the academic years of 2010-2011, 2011-2012 and 2012-2013 we began to implement both experimentally in the School of Computer Science at the Complutense University of Madrid. For both the educational strategy and the software system that supports it, we conducted studies involving both teachers and students. Thus, regarding studies with teachers:

- During the 2010-2011 academic year a study was performed with teachers in the Language Processors course in order to assess the initial version of the *Evaluators* system's authoring tool, based on a constructivist approach. The results shown that the initial authoring tool was not suitable for the group of teachers interviewed. This results was the seed of the current version of the authoring tool [135] [142], based on a generative approach. Later this assessment study was repeated in mid 2010-2011 academic year with the new version of the tool, where positive results were obtained [135].
- Also during the 2010-2011 course another study was conducted with teachers of the course to evaluate the *Evaluators* analysis tool. The observations obtained were not positive, which led to a rethinking of the tool and to its current version [135] [142], whose subsequent evaluation (during the academic year 2011-2012) was positive [135].

Concerning studies carried out with students:

- During the 2010-2011 academic year a short-term experience was performed with students where we evaluated the educational effectiveness of the strategy, the simulators based on serious games, and the degree of student satisfaction. In this experience, it became clear that the educational effectiveness of the strategy and the serious games was similar to that obtained with traditional teaching methods, but the satisfaction among students was greater, thus increasing the motivation of students to study the subjects treated more actively than the traditional method [135] [137].

- During the academic year of 2011-2012, short-term experiences oriented to assessing both the level of student satisfaction and the educational effectiveness of the strategy and the generated serious games were repeated. The study results were similar to those obtained during the 2010-2011 academic year [135].
- Apart from conducting short-term assessment experiences during the 2010-2011 and 2011-2012 courses, all students followed the strategy and were able to use the software support during the whole course. This allowed us to make a long-term study comparing the results obtained during these two academic years and previous years. This study showed that the use of the strategy and the system based on serious games had a positive effect on student learning compared to other academic years where none of them were used [135] [141].
- Finally, during the academic year 2012-2013, an assessment experience oriented to measure the degree of satisfaction of the students and the educational effectiveness of the strategy and supporting software that creates interactive visualization-based simulators was performed. The students showed a high degree of satisfaction with the use of simulators, considering them suitable for learning. Moreover, the comparative study to measure educational effectiveness showed promising results with respect to the traditional teaching method of these concepts [135] [139].

Thus, the results of teachers and students show that both the educational strategy, as the accompanying software system (with two possible types of simulators generated), are suitable for teaching concepts related to the semantic evaluation process that characterizes attribute grammars. In addition, a positive reception by teachers and students from both the educational strategy and the software system is also observed.

7.4.1.4 Formulation the process model

To end the achieving of the objective 1, in this thesis, the underlying development model of *Evaluators* has been abstracted to facilitate the construction of similar systems. The resulting process model provides, facing the general objective of this thesis, a methodology to produce similar software systems to *Evaluators*, but focused on other specification formalisms.

The process model is a strategy for the development of educational systems building simulations based on a generative approach. Thus, the systems developed are able to automatically generate educational simulations from collections of exercises. The model organizes the work of teachers, developers and students in different tasks. The first tasks proposed for the approach focuses on establishing the type of exercises and educational simulations that teachers and developers want for the system. Then, developers create different software tools that compound the system from the specifications agreed in the previous task: authoring tool, generator simulator and analysis tool. In later tasks, teachers use the tools developed to create exercises batteries (by authoring tool) and the corresponding simulators (using the simulator generator). Then students use the generated educational simulators to solve the problems posed by teachers.

Generated simulators will be able to store the student activity in log files, which then teachers can analyze (using the analysis tool) to evaluate solutions of the students.

In parallel to the production process explained above, there is another process oriented to the evaluation of the products created that complete the development strategy. The objective of these assessments is to identify problems and areas of improvement of software products designed to suit the needs of end users of these products, students and teachers. So, when a product is created for the main workflow of the proposed strategy, in turn an *evaluation instrument* is also created where key aspects of the product created to evaluate are specified along the evaluation method to be used. Later, when this product is used, in turn the evaluation instrument associated with it will be applied by an actor involved (student or teacher), recording in a *evaluation report* the assessment results obtained. These reports are analyzed and can cause reactivation of the different tasks of the main production flow. These reactivations are intended to enhance products previously created with the information gathered in the evaluation report.

In [137] we have described a draft of the process model focused on the development of serious games. In [138] the current version thereof focused on the generation of interactive simulations is described. In [135] we provided details on how this process model emerges from the development experience from *Evaluators*.

7.4.1.5 Conclusions

Throughout this section we have contextualized, based on publications, the different tasks undertaken to fulfill objective 1, that is, to provide, support by software tools and evaluate an educational strategy to improve the teaching-learning of basic concepts of attribute grammars in early stages of instruction. Thus, first the educational strategy was conceived to be focused on the most problematic aspect of the formalism: its computational model, driven by the dependencies between attributes. This requires a problem-based approach, designing a specific type of exercise that lets students comprehend these aspects through experimentation. As a result, the strategy was supported by a software system called *Evaluators* that can generate serious games and interactive simulations.

The experiences of evaluation, carried out with teachers and students, to validate the adequacy and effectiveness of the educational strategy and the supporting software, have highlighted the good reception among teachers and students of the strategy and supporting software against more traditional teaching methods. These experiences have also shown improvements regarding the educational effectiveness of the methods used.

Finally, the approach followed for the construction of the software system is also able to be abstracted satisfactorily by a process model applicable to other scenarios. The resulting process model can be used to support strategies driven by problems through a generative approach modulated by assessments provided by teachers and students. In the framework of this thesis, the aim of this model is to

extrapolate the proposals to other scenarios involving different formalisms, and even to other teaching-learning contexts.

7.4.2 Strategy for the implementation of attribute grammars as translation schemata

The development strategy designed to facilitate the development of language processors allows students to obtain working implementations, as translation schemata, from specifications based on arbitrary non-circular attribute grammars. To do this so, the strategy relies on the use of a software library that facilitates the task together with conventional language processors development tools. Finally, in order to validate both the development strategy and the supporting software, we have conducted various studies and experiences in a Language Processor course.

Subsection 7.4.2.1 contextualizes the development strategy. Subsection 7.4.2.2 contextualizes the development of the library. Subsection 7.4.2.3, finally, contextualizes the evaluation carried out.

7.4.2.1 Formulation of the development strategy

In [143] [144] we described the formulated development strategy. This strategy is capable of obtaining the implementation of a language processor, by using processor generation tools based on translation schemata, from its specification as an attribute grammar. Specifically, the strategy is based on the following processes for the implementation of the specified language processor:

- First, the syntax rules for the input specification, described in terms of an attribute grammar, are transcribed in readable terms into the notation supported by a generation tool based on translation schemata. Specifically, [144] the strategy developed advocates the use of tools for generating bottom-up parsers/translators, as CUP, although in [143] the strategy is also generalized for top-down parser / translator generators, like ANTLR or JavaCC.
- Later on, the semantic equations will be transcribed as semantic actions within the translation scheme. To do this, the semantic equations should be described in terms of dependencies between attributes that are derived from them, and the method needed to perform semantic evaluation.

The resulting translators couple the process of semantic evaluation and the analysis process in a seamless way. Likewise, the evaluation process can be directed by the dependencies, or proceed on demand. In any case, all these aspects are transparent to the students who perform the encoding. Thus, with the strategy presented, students are able to get implementations of the language processors that are designed directly from their specifications in terms of arbitrary non-circular attribute grammars. Aside from being able to validate the design in an experimental way, students are able to identify the correspondences between the translation scheme and the original specification described by the formalism of attribute grammars. Also, as described in [143][144], students can also refine the implementation through successive optimizations, and they can always trace the relationships with the original specification.

7.4.2.2 Development of the supporting software

The development strategy proposed above requires software support. Specifically, as described in [136], we have developed a software library called *EvLib* which provides the necessary methods to specify the semantic equations as semantic actions within the translation scheme obtained from the direct implementation of the development strategy. In such semantic actions, the dependencies between attributes and the evaluation method are specified using the library. Those dependencies and evaluation method are derived from the corresponding semantic equation. In addition, *EvLib* provides a semantic evaluation engine for the generated processors capable of executing semantic calculations properly, according to the computational model of the attribute grammar formalism.

7.4.2.3 Validation of the strategy and the supporting software

To validate the adequacy of the strategy and its supporting software, we conducted different experiences between teachers and students during the 2012-2013 Language Processors course in the Complutense University of Madrid. These results are described in [136]. First, the assessment results obtained from teachers and students were positive. Although the strategy is considered by students as appropriate to develop their processors, they consider insufficient or misleading the debugging information provided by the generation processor tools and *EvLib* library. On the other hand, in [136], the results of educational effectiveness are compared to the ones obtained from the traditional implementation strategies used in the course. It can be seen that students who used the development strategy that preserves the original design have performed better than students who used the traditional strategy.

With these results, we can conclude that the development strategy, as well as the educational software, are suitable for students of a Language Processors course, validating both assets for their main purpose: facilitate the development of language processors to students.

7.4.2.4 Conclusions

In conclusion, the proposed development strategy and the *EvLib* library support, provides students with a framework that allows them to develop their own processors from the specification in terms of a attribute grammar without using specialized tools (only conventional development tools based on translation schemata). With the proposed development approach students get the tools to develop (and validate) their language processors directly from its specification and its subsequent implementation, as evidenced by the results obtained in the different experiences presented in [136]. Also, as discussed in [143] [144], the approach also allows a strategy based on evolution of prototypes, where the initial prototype obtained can be transformed into a final implementation by the successive application of refinements. In this evolutionary process students always can trace each intermediate artifact back to the original specification.

7.4.3 Educational language processor development environment based on attribute grammars

Objective 3 raised in this thesis has been satisfied building a platform for the development of language processors based on the formalism of attribute grammars that facilitates the understanding of this formalism by using strategies employed to achieve the objective 1, and, that in turn, generates implementations for processors based on the processes proposed to achieve the objective 2. In addition, we have driven an empirical validation of the appropriateness and usefulness of the educational system developed for use in a course of Language Processors.

Subsection 7.4.3.1 contextualizes the results concerning the development of the platform. Subsection 7.4.3.2 contextualizes the results for evaluation.

7.4.3.1 Development of a software platform for the creation of language processors

The development platform created, and described in [140], is intended for the automatic generation of language processors from its specification in terms of an attribute grammar. The tool provides a full attribute grammar editor to allow students to specify their language processors using such formalism. Moreover, the tool has a powerful visual debugger that, given a specified processor and a sentence for that processor, displays the decorated parsing tree of such a sentence, and emulates the evaluation process of the values of each attribute instances decorating that tree. The debugger supports the comprehension strategy developed to achieve the objective 1. In this way, students will be able to have a better understanding of the internal behavior of their own language processors. Finally, the platform generates language processors from its specification as attribute grammar automatically applying the development strategy raised to achieve the objective 2. Thus, the generated implementations remain readable to students due to the correspondence with the original specification.

7.4.3.2 System validation

System validation was carried out during a research stay at the Faculty of Computer Science, University of Minho (Portugal), in the group of Prof. Pedro Rangel Henriques. This stay was made during the course 2013-2014, and allowed us to carry out an experience with students of Language Processors course where we had the opportunity to measure the degree of satisfaction of the students with the environment and validate the perceived educational value. In [140] the results are recorded, proving that the tool has been successful among students, who consider it useful for specifying, implementing and debugging their own language processors. Despite possible improvements and problems that students encountered during the experiment carried out, both the attribute grammars editor and the visual debugger were assessed positively.

7.4.3.3 Conclusions

In conclusion, the development environment created can satisfy objective 3 providing tools to help students develop their language processors from their specifications. The development environment created provides students with the necessary tools to understand and refine their own specifications as attribute grammars thanks to the editor and visual debugger. In addition, the tool automatically performs the implementation steps that students should address to obtain the translators from their specifications following the development strategy proposed in the previous section, allowing students to identify the correspondence between the specification introduced and the implementation obtained.

7.5 Conclusions and future work

In previous sections, we have presented different problems and educational needs in the field of teaching Language Processors. For those needs, we have proposed different strategies and software tools whose main objective is to improve the educational experience, for teachers and students, of concepts in a Language Processors course, and more specifically of concepts related to the attribute grammar formalism. As detailed in the previous section, the results obtained when covering the objectives proposed in this thesis are described in different papers included herein. Thus, this final section concludes this PhD dissertation summarizing the main conclusions and presenting some lines of future work.

7.5.1 Main contributions

In this section, we will present the main contributions made in this thesis. More specifically, and based on the discussion in the preceding sections, we highlight the following contributions:

- Proposal of an educational strategy for the comprehension of specifications based on attribute grammars, which is supported by a software system able of generating different types of educational simulators.
- Formulation of a strategy for the implementation of arbitrary non-circular attribute grammars as translation schemata.
- Creation of a language processor development environment acting on attribute grammar-based specifications that facilitates the understanding of the specifications and their implementation as translation schemata.

The following subsections describe in detail each contribution.

7.5.1.1 Educational strategy for the comprehension of attribute grammar-based specifications

In this thesis, we have formulated an educational strategy to improve the teaching-learning of basic concepts of the formalism of attribute grammars in early stages of learning within a Language Processors course. This strategy focuses on teaching concepts related to semantic evaluation using exercise batteries related to

those concepts. These exercises will propose students a language processing problem providing the following information:

- Informal description of a language processor.
- Formal description of the language processor specified by the attribute grammar formalism.
- A language sentence to process.
- The decorated syntax tree derived from the processing of the sentence with the attribute grammar provided.

With this information, the solution that students have to provide must be a correct attribute evaluation order, according to the given grammar. Note that there may be different valid evaluation orders for the same exercise. Furthermore, to support the strategy, we have created a software system called *Evaluators*, characterized by:

- An authoring tool to facilitate teachers to produce exercises allowing them to enter the description of the language and sentence required to generate all the information needed to define each exercise of the battery.
- Generators that are able to build educational simulators based on serious games and interactive visualizations from these exercise batteries. There will be in these generated simulators where students solve the exercises proposed by teachers. Both types of simulators present the information for each exercise as follows:
    - Simulators based on serious games present to the student a maze based on the structure of the syntax tree where there are boxes that represent the attributes. The aim is to move objects, representing the semantic value of each attribute to other boxes in order to create new objects. In this way, the student implicitly provides an evaluation order of the attributes that decorate the syntax tree.
    - The interactive simulators based on visualizations have the same dynamic that the simulators based on serious games but the information is presented by visual representations very close to the formal representations used in classroom. The simulator shows a representation of the decorated syntax tree where students must move the values of the attributes to compute. And it is in this process where the student specifies an evaluation order.
- Finally, the analysis tool lets teachers evaluate the solutions proposed by the students on the simulators generated and monitor their progress. This tool displays, on a syntax tree associated to each of exercise in the battery, the actions carried out by the students during the simulation.

On the other hand, and in order to validate both the educational strategy and software system that will support it, *Evaluators*, we have conducted different experiences between teachers and students of Language Processors courses. In these experiences, it has become clear how the assessments between teachers and students are good, both concerning the strategy formulated and the software that supports it. In addition, the studies described in the previous section concerning the educational effectiveness of the strategy and the software have also demonstrated the utility of these for students and their learning, both to the short-

term (with specific experiences) and to the long term (with continuous use over different academic years).

Finally, all the experience gained during the development of the educational strategy and software system that supports it has been summarized in a process model for the generative production of educational systems based on exercise batteries and driven by teachers and students assessments. This process model's main objective is to abstract the good practices identified during the development of *Evaluators* and apply them to other education fields in general, and specifically in the course of Language Processors for other specification formalisms.

7.5.1.2 Strategy for the implementation of attribute grammars as translation schemata

In order to help students of a Language Processors course to identify issues related to their own specifications, in this thesis we have proposed a strategy to develop language processors as translation schemata from arbitrary non-circular attribute grammars. This strategy allows students to produce translation schemata that preserve the original specification in attribute grammar form, making it easy for students to debug errors and understand the relationship between a specification of a processor by an attribute grammars and its final implementation because there is a direct correspondence between the original specification and the implementation produced by this strategy.

The educational development strategy aims directly to involve the student in the development process. Although the strategy is easily automated, it is important to involve the students in the process of coding to make them aware of the relationship between original attribute grammars and the subsequent implementation, and thus improve their learning and understanding. The proposed coding process consists of two main stages:

- First, the syntax rules of the original attribute grammar are transcribed as syntax rules of a translation schemata, processable by a language processors generation tool. Specifically, the strategy proposes the use of bottom-up parser/translator generators, as CUP, but is also applicable, with further development, to top-down parser/translator generators.
- Later on, semantic equations must be encoded as semantic actions within the translation scheme. Therefore, the strategy requires students to transcribe the semantic equations in terms of dependencies between attributes that are derived from them, and the required evaluation method.

In order to make the calculation process of the generated translation schemata-based implementations faithful to the semantic evaluation process of the attribute grammars, we have developed a software library, called *EvLib*, to be used to specify the evaluations as semantic actions in the generated translation schemata. Thus, the processors implemented preserve the syntactic and semantic aspects of attribute grammars.

Finally, we have carried out different experiences with teachers and students in a Language Processors course to measure their satisfaction degree. and to evaluate the educational effectiveness of the strategy. The development approach was successful among teachers and students who found it useful for the development

of their own language processors. On the other hand, educational efficacy observed demonstrates the utility of the approach against other more conventional and more frequently used development strategies in a Language Processors course.

7.5.1.3 Educational language processor development environment based on attribute grammars

As an integration result of this thesis, it has also been implemented a language processors development environment based on specifications in the form of attribute grammars that fulfills two main features:

- It facilitates the understanding of the specifications by using a sophisticated visual debugger. This debugger displays, in an appealing fashion, the evaluation process carried out by processors according to attribute grammars that specify the processors and the sentences of the languages defined by these grammars, by drawing the decorated syntax trees and by animating the evaluation process on that tree, showing how the semantic values move on that tree. This approach is based on the works related to *Evaluators*.
- It generates implementations based on translation schemata following the coding patterns proposed in the previous point referred to the implementation strategy. In this way, students will be able to identify implementations generated with the original specifications described in the formalism of attribute grammars. This feature is based on the strategy for the implementation of non-circular attribute grammars as translation schemata and makes use of the *EvLib* library.

Finally, we measured the perceived usefulness of the environment with teachers and students in a Language Processors course. The results show that both, teachers and students, consider the development environment as a useful and a fully highly suitable tool for a Language Processors course.

7.5.2 Future work

Finally, this section concludes this thesis outlining the most promising lines of future work, derived from the proposals presented. These lines of future work are listed below:

- Further development of *Evaluators*.
- Exploration of the applicability of the process model for the development of educational systems in other domains.
- Experiencing the long term effects on student learning of the language processors development strategy.
- *EvDebugger* further development.
- Freeware distribution of the tools and software

The following subsections motivate each of these lines of future work and research.

7.5.2.1 Further development of *Evaluators*

In this research, various improvements are proposed, related to software tools that compound the educational system *Evaluators*, which supports an educational strategy to improve learning of the attribute grammar formalism. Also, we propose to conduct more extensive studies on the educational strategy, the system tools and the software simulators created, with the aim of identifying potential enhancement aspects.

7.5.2.1.1 *Evaluators* educational system improvements

Improvements on the authoring tool are summarized in improving the attribute grammars editor with features similar to a development environment, where tools provide auto-completion and auto-correction capabilities. Also, being described language processors, it would also be interesting to present more sophisticated debugging messages to help teachers identify possible errors in their specifications.

On the other hand, improvements on the analysis tool focus on providing functionality for mass and automatic solution processing. This feature will detect the most common errors made in the exercises that teachers can identify and plan educational activities to reinforce concepts involved in such failures.

Finally, both types of generated simulators are also being the focus of possible improvements. Specifically, a possible future enhancement involving both of them is related to the feedback presented. Some students, from the experiences carried out, exposed that the identification of mistakes made during the simulation were sometimes difficult to interpret. Moreover, teachers interviewed also pointed to this lack (in particular teachers and researchers from the Computer Science School at the University of Utrecht, Netherlands, which pointed to this problem during a short visit by the author of this thesis to the research group of Prof. Johan Heuring, in the context of the Spanish National R&d&i Project TIN2010-21288-C02-01, " A Generative Approach to the Development of Learning Object Production and Deployment Tools in the Virtual Campus"). Therefore, we propose as future work, the incorporation to of a rule-based intelligent tutor to the generated simulators in order to be able to make useful suggestions to guide the resolution of exercises for each fault that the student could make during simulation. This line of research may be conducted as part of a future collaboration with the aforementioned researchers from the University of Utrecht.

7.5.2.1.2 Comparative study of different *Evaluators* simulators

This line of research raises as a need to identify the strengths and weaknesses of the two types of simulators generated with *Evaluators*. In this thesis, we have shown evaluation results of each of the simulators against to the traditional method for solving the exercises in the education strategy formulated. From these experiences, it has been observed that satisfaction among teachers and students facing these simulators was good, as with the educational effectiveness of both simulators compared to the traditional method. However, we believe that a comparative study between the two types of simulators will give us a different

view of students and teachers, where we can identify strengths and weaknesses in both types of simulators. In this way, we will be able to plan future improvements in both simulators that will make them more effective and appealing.

7.5.2.2 Application of the process model to the development of educational systems in other domains

The future line of work presented here has been briefly mentioned in this thesis, as it is one of the reasons why the process model for the development of educational software systems was created.

The immediate objective is to apply the process model to develop an educational software system to facilitate the acquisition of basic concepts of the translation schemata formalism. However, this process model may be applicable to other fields of Computer Science education as algorithms, data structures or formal logic.

We have even planned the implantation of the process model in other areas, apart from computer science, as teaching literary analysis, classification corpus of historical documents or leadership in education. The possibility of working in other educational settings can lead to substantial improvements in the process model proposed by interacting with teachers and students with different educational needs and perspectives to those found in the field of Computer Science. This will undoubtedly enrich the process model, becoming a more versatile one. These developments will be carried out within research projects in the digital humanities and e-learning domain that are currently active in the ILSA research group: the HUM14_251 "Unified Management Model of Digital Collections with Reconfigurable Structures: Application to the creation of specialized digital libraries for Research and Teaching" project, awarded by the BBVA Foundation in its 2014 Call for Research Project Proposals, and the TIN2014-52010-R "Repositories with Dynamic Reconfigurability in Humanities" project, issued in the 2014 Call of the Spanish National Research Program oriented to the Societal Challenges. In the framework of these projects, we could plan and implement the development of educational software targeted on issues of these projects through direct application of the software development approach proposed in this thesis.

7.5.2.3 Long-term experiences with the language processor development strategy

In this thesis, there have been different empirical results relating to the implementation strategy of non-circular attribute grammars as translation schemata. These results were originated by several short-term experiences with students and teachers planned as isolated activities within a Language Processors course. Although the results of these experiences were satisfactory, it is necessary to observe the educational effect of the development strategy for a full course of Language Processors. Therefore, this line of future work arises as a study of the evaluation of educational effectiveness of the development strategy formulated where students will develop language processors during the course using that

strategy. In this way, we would obtain enough empirical evidence to assess the long-term educational effects of the development strategy within a Language Processors course.

7.5.2.4 Further development of *EvDebugger*

This line of research focuses on two main areas: improving *EvDebugger*, the development environment based on attribute grammars, and perform studies of educational effectiveness of this tool.

7.5.2.4.1 Improvement of the *EvDebugger* environment

The *EvDebugger* version presented in this thesis, is an early version that shows some deficiencies of use and accessibility that will be corrected in future releases. One of the most significant improvements, which arises as a line of future work, is to provide more sophisticated debug messages. These messages should provide useful information for the language processors specification during compilation and during visual debugging. On the other hand, it is necessary to provide to the visual debugger with a method for selecting breakpoints during the debugging process. In this way, users can access specific points of the calculation process of semantic attributes without the need to emulate the whole process. Finally, we are considering to extending the kind of processable attribute grammars and to include the ability to process ambiguous grammars.

All these improvements stem from the short stay made in the University of Minho (Braga, Portugal), and their development will be carried out as future collaborations with the host research group during the short stay: gEPL - Grupo de Especificaçao e Processamento de Linguagens, directed by Prof. Pedro Rangel Henriques.

7.5.2.4.2 Studies of educational effectiveness of *EvDebugger*

Finally, this research line proposes to measure the effectiveness of *EvDebugger* as a supporting tool for a Language Processor course. In this thesis, we have presented results of an experience assessment conducted between teachers and students in a course of Language Processors in the aforementioned University of Minho. These results are promising, since *EvDebugger* has achieved success among teachers and students of the course. Therefore, this line of work proposes to conduct separate studies to measure the educational effectiveness of *EvDebugger*, both at the short and the long term, to obtain empirical evidence of the educational potential of this tool. Furthermore, from these studies, we hope to obtain relevant information that may lead to transform *EvDebugger* into a more effective and engaging educational software tool for students.

7.5.2.5 Freeware distribution of the tools and software

Finally, as part of the dissemination and technology transfer activities associated with this thesis, we are planning to distribute as freeware the tools and software developed (*Evaluators*, *EvLib*, *EvDebugger*). The aim of this effor will be to

encourage their wide use among instructors and students of subjects related to the design and implementation of computer languages.

Referencias

[1] Ablas, H. Attribute Evaluation Methods. En Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 48-113. 1991.

[2] ACM/IEEE. Computer Science Curriculum 2008: An Interim Revision of CS 2001. 2008.

[3] Adams, W.K., Reid, S., LeMaster, R., McKagan, S.B., Perkins, K.K., Dubson, M., Wieman, C.E. A Study of Educational Simulations Part I - Engagement and Learning. Journal of Interactive Learning Research, 19(3): 397-419. 2008.

[4] Aho, A.V. Teaching the Compilers Course. ACM SIGCSE Bulletin, 40(4): 6-8. 2008.

[5] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools (Second Edition). Addison-Wesley. 2007.

[6] Aiken, A. Cool: A portable project for teaching compiler construction. ACM SIGPLAN Notices, 31(7): 19-24. 1996.

[7] Akker, R., Melichar, B., Tarhio, J. Attribute Evaluation and Parsing. En Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 187-214. 1991.

[8] Akker, R., Melichar, B., Tarhio, J.: The Hierarchy of LR-attributed grammars. En WAGA'90: Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications, Lecture Notes in Computer Science 461, Springer, 13-28. 1990.

[9] Allen, W. C. Overview and evolution of the ADDIE training system. Advances in Developing Human Resources, 8(4): 430-441. 2006.

[10] Allenstein, B., Yost, A., Wagner, P., Morrison, J. A query simulation system to illustrate database query execution. En SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education, 493-497. 2008.

[11] Almeida-Martínez, F. J., Urquiza-Fuentes, J., Velázquez-Iturbide, J. Á. Vast: a visualization-based educational tool for language processors courses. ACM SIGCSE Bulletin, 41(3): 342-342. 2009.

[12] Almeida-Martínez, F. J., Urquiza-Fuentes, J., Velázquez-Iturbide, J. Á. VAST: Visualization of abstract syntax trees within language processors courses. En SOFTVIS'08: Proceedings of the 4th ACM symposium on Software visualization, 209-210. 2008.

[13] Amory, A., Naicker, K., Vincent, J., Adams, C. The Use of Computer Games as an Educational Tool: Identification of Appropriate Game Types and Game Elements. British Journal of Educational Technology, 30(4): 311-321. 1999.

[14] Appel, A.W. Modern Compiler Implementation in Java. Cambridge University Press. 2002.

[15] Arbab, B. Compiling Circular Attribute Grammars into Prolog. IBM Journal of Research and Development, 30(3), 294-309. 1986.

[16] Augusteijn, L. The Elegant Compiler Generator System. En WAGA'90: Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications, Lecture Notes in Computer Science 461, Springer, 238-254. 1990.

[17] Avgeriou, P., Retalis, S., Papaspyrou, N. Modeling Learning Technology Systems as Business Systems. Journal Software and Systems Modeling, 2(2), 120-133. 2003.

[18] Aycock, J. The Art of Compiler Construction Projects. ACM SIGPLAN Notices, 38: 28-32. 2003.

[19] Baldwin, D. A Compiler for Teaching about Compilers. En SIGCSE'03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, 220-223. 2003.

[20] Bednarik, R., Gerdt, P., Miraftabi, R., Tukiainen, M. Development of the TUP Model - Evaluating Educational Software. En ICALT'04: Proceedings of the 4th IEEE International Conference on Advanced Learning Technologies, 699-701. 2004

[21] Berson, M.J. Effectiveness of Computer Technology in the Social Studies: A Review of the Literature. Journal of Research on Computing in Education, 28(4): 486-499. 1996.

[22] Bischoff, K.M. Design, Implementation, Use and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex and C. TR #92-31, Dp. Of Computer Science, Iowa State University. 1992

[23] Blanton, E., Lessa, D., Arora, P., Ziarek, L., Jayaraman, B. JI. FI: Visual test and debug queries for hard real-time. Concurrency and Computation: Practice and Experience, 26(14): 2456–2487. 2013

[24] Bloom, B., Engelhart, M., Furst, E., Hill, W., Krathwohl, D.R. Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook I, The cognitive domain. Longman Group. 1969.

[25] Bochmann, G.V. Semantic Evaluation from Left to Right. Communications of the ACM, 19(2): 55-62. 1976

[26] Bovet, J., Parr, T. ANTLRWorks: an ANTLR Grammar Development Environment. Software: Practice and Experience, 38(12): 1305-1332. 2008.

[27] Boyle, T. Design Principles for Authoring Dynamic, Reusable Learning Objects. Australian Journal of Educational Technology, 19(1): 46-58. 2003.

[28] Bradley C., Boyle T. The Design, Development and Use of Multimedia Learning Objects. Journal of Educational Multimedia and Hypermedia, Special Edition on Learning Objects, 13(4): 371-389. 2004

[29] Brand, M.G.J v.d., Deursen, A, v., Heering, J., Jong, H.A.d., Jonge, M.d., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, JJ., Visser, E., Visser, J. The Asf +Sdf Meta-environment: A Component-Based Language Development Environment. En CC'01: Proceedings of the 10th International Conference on Compiler Construction, Lecture Notes in Computer Science, 2027, Springer, 365-370. 2001.

[30] Bridgeman, S., Goodrich, M. T., Kobourov, S. G., Tamassia, R. PILOT: An interactive tool for learning and grading. ACM SIGCSE Bulletin, 32(1): 139-143. 2000.

[31] Cameron, M., Garcia-De-La-Banda, M., Marriott, K., Moulder, P. Vimer: a Visual Debugger for Mercury. En PPDP'03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming, 56-66. 2003.

[32] Castro-Schez, J.J., Redondo, M.A., Gallardo, J. Jurado, F. Designing and Developing Software for Educative Virtual Laboratories with Language Processing Techniques: Lessons Learned in Practical Experiments. Journal of Research and Practice of Information Technology, 44(3): 289-308. 2012.

[33] Cervelle, J., Forax, R., Roussel, G. Tatoo: an Innovative Parser Generator. En PPPJ'06: 4th International Symposium on Principles and Practice of Programming in Java, 13-20. 2006.

[34] Copeland, T. Generating Parsers with JavaCC. Alexandria: Centennial Books. 2007.

[35] Costa, A. P., Loureiro, M. J., Reis, L. P. Development Methodologies for Educational Software: the Practical Case of Courseware SERe. En EDULEARN'09: Proceedings of Conference on Education and New Learning Technologies, 5816-5825. 2009.

[36] Czarnecki, K., Eisenecker, U. Generative Programming: Methods, Techniques and Applications. Addison Wesley. 2000.

[37] Damaševičius, R., Štuikys, V. On the Technological Aspects of Generative Learning Object Development. En ISSEP'08: 3rd International Conference on Informatics in Secondary Schools - Evolution and Perspectives. 337-348. 2008.

[38] De Freitas, S., Jarvis, S. Towards a Development Approach for Serious Games. En Connolly, T., Stansfield, M.,Boyle, L (Eds): Games-based Learning Advancements for Multi-Sensory Human-Computer Interfaces: Techniques and Effective Practices, 215-231. IGI Global. 2008.

[39] Demaille, A. Making Compiler Construction Projects Relevant to Core Curriculums. En ITiCSE'05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, 266-270. 2005.

[40] Desherm, H. L., McFall, R.L., Uti, N. Animation of Java linked lists. En SIGCSE'02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, 53-57. 2002.

[41] Deshpande, A.A., Huang, S.H. Simulation Games in Engineering Education: A State-of-the-Art Review. Computer Applications in Engineering Education, 19(3): 399-410. 2011

[42] Diehl, S., Kerren, A., Weller, T. Visual Exploration of Generation Algorithms for Finite Automata on the Web. En CCIA'01: Revised Papers of from the 5th International Conference on Implementation and Application of Automata, Lecture Notes in Computer Science 2088, 327-328. 2001

[43] Díez, D., Fernández, C., Dodero, J. A Systems Engineering Analysis Method for the Development of Reusable Computer-Supported Learning Systems. Interdisciplinary Journal of E-Learning and Learning Objects, 4(1): 243-257. 2008.

[44] Donnelly, C., Stallman, R. Bison. The YACC-compatible Parser Generator. Free Software Foundation. 2004.

[45] Eagle, M., Barnes, T. Wu's castle: Teaching Arrays and Loops in a Game. En ITiCSE '08: Proceedings of the 13th annual Conference on Innovation and Technology in Computer Science Education, 245-249.2008.

[46] Eagle, M., Barnes, T. Experimental Evaluation of an Educational Game for Improved Learning in Introductory Computing. En SIGCSE '09: Proceedings of the 40th ACM Technical Symposium on Computer Science Education, 321-325. 2009.

[47] Ekman, T., Hedin, G. The JastAdd system - Modular Extensible Compiler Construction. Science of Computer Programming, 69(1-3), 14-26. 2007.

[48] Elsworth, E.F.. The msl Compiler Writing Project. ACM SIGCSE Bulletin, 24(2): 41-44. 1992.

[49] Eysholdt, M., Behrens, H. Xtext: Implement your Language Faster than the Quick and Dirty Way. En OOPSLA'10: Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications, 307-309. 2010.

[50] Fowler M. Domain-Specific Languages. Addisson-Wesley. 2010.

[51] Friesen, K., Schmitz, H. Managing the Development of an E-learning Product–Applying Software Engineering Techniques in an Academic Environment. En NL'02: Proceedings of the World Congress of Networked Learning in a Global Environment, 40-46. 2002.

[52] Gagnon, E.M., Hendren, L.J. SableCC, an Object-Oriented Compiler Framework. En TOOLS'98: International Conference on Technology of Object-Oriented Languages, 140-154. 1998.

[53] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995.

[54] García-Osorio, C.; Gomez-Palacios, C.; Garcia-Pedrajas, N. A Tool for Teaching LL and LR Parsing Algorithms. In ITiCSE'08: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, 317-317. 2008.

[55] García-Pacheco, I., García-Matías, J. A Methodology Based on Effective Practices to Develop Educational Software. Computación y Sistemas, 11(4): 313-332. 2008.

[56] Garris, R., Ahlers, R., Driskell, J.E. Games, Motivation and Learning: A Research and Practice Model. Simulation & Gaming, 33(4): 441-467. 2002.

[57] Gee, J.P. What Video Games have to Teach Us about Learning and Literacy. Palgrave Macmillan. 2003.

[58] Gómez-Martín, M.A. Arquitectura y Metodología para el Desarrollo de Sistemas Educativos basados en Videojuegos. Tesis doctoral. 2007.

[59] Gómez-Martín, M.A., Gómez-Martín, P.P., González-Calero, P.A. Dynamic Binding is the Name of the Game. En ICEC'06: Proceedings of International Conference on Entertainment Computing, 229-232. 2006.

[60] Gómez-Martín, M.A., Gómez-Martín, P.P., González-Calero, P.A. Game-Driven Intelligent Tutoring Systems. En ICE'04: Proceedings of the International Conference on Entertainment Computing, 108-113. 2004.

[61] González, C., Toledo, P., Muñoz, V., Noda, M.A., Bruno, A., Moreno, L. Inclusive educational software design with agile approach. En TEEM'13: Proceedings of 1st International Conference on Technological Ecosystem for Enhancing Multiculturality, 149-155. 2013

[62] GRAPPA. A Java Graph Package. http://www2.research.att.com/~john/Grappa/

[63] Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M. Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM, 35: 121-131. 1992.

[64] Grinder, M.T. A Preliminary Empirical Evaluation of the Effectiveness of a Finite State Automaton Animator. En SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education, 157-161. 2003.

[65] Grinder, M.T. Animating Automata: a Cross-Platform Program for Teaching Finite Automata. En SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education, 63-67.2002.

[66] Griswold, W.G. Teaching Software Engineering in a Compiler Project Course. ACM Journal of Educational Resources in Computing, 2(4): 18pp. 2002.

[67] Hambach, S., Martens, A. ROME: Systems Engineering in Technology Enhanced Learning. En ICALT'08: Proceedings of the 8th IEEE International Conference on Advanced Learning Technologies, 733-734. 2008.

[68] Hedin, G. An Object-Oriented Notation for Attribute Grammars. En ECOOP'89: Proceedings of the 3rd European Conference on Object-Oriented Programming, 329-345. 1989.

[69] Henk A. Attribute Evaluation Methods, En Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 48-113. 1991

[70] Henriques, P.R., Varanda-Pereira, M.J., Mernik, M., Lenic, M., Gray, J.G., Wu, H. Automatic Generation of Language-Based Tools using the LISA System. IEEE Proceedings – Software, 152(2): 54-69. 2005.

[71] Hübscher-Younger, T., Narayanan, N. H. Dancing Hamsters and Marble Statues: Characterizing Student Visualizations of Algorithms. En SOFTVIS'03: Proceedings of the 2003 ACM Symposium on Software visualization, 95-104. 2003.

[72] Hudson, S. E., Flannery, F., Ananian, C. S. CUP LALR Parser Generator for Java. 1999. https://www.cs.princeton.edu/~appel/modern/java/CUP/

[73] Hundhausen, C. D., Douglas, S. A., Stasko, J. T. A Meta-Study of Algorithm Visualization Effectiveness. Journal of Visual Languages & Computing, 13(3): 259-290. 2002.

[74] Jalili, F. A General Linear-Time Evaluator for Attribute Grammars. ACM SIGPLAN Notices, 18(9): 35-44. 1983.

[75] Jiménez-Díaz, G., Gómez-Albarrán, M., González-Calero, P. A. Pass the Ball: Game-based Learning of Software Design. En ICEC'07: Proceedings of the International Conference on Entertainment Computing, 49-54. 2007.

[76] Jiménez-Díaz, G., González-Calero, P.A., Gómez-Albarrán, M. Role Play Virtual Worlds for Teaching Object Oriented Design: The ViRPlay Development Experience. Software: Practice and Experience, 42(2): 235-253. 2012.

[77] Jodar-Reyes, J.F; Revelles-Moreno, J. SEFALAS: Software para la Enseñanza de las Fases de Análisis Léxico y Análisis Sintáctico. http://lsi.ugr.es/plweb/static/software.html

[78] Jones, L.G. Efficient Evaluation of Circular Attribute Grammars. ACM Transactions on Programming Languages and Systems, 12(3), 429-462. 1990.

[79] Jong, T., Van Joolingen, W. R. Scientific discovery learning with computer simulations of conceptual domains. Review of educational research, 68(2), 179-201. 1998.

[80] Jourdan, M., Parigot, D. Internals and Externals of the FNC-2 Attribute Grammar System. En Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 485-504. 1991.

[81] Kaplan, A., Shoup, D. CUPV. A Visualization Tool for Generated Parsers. In SIGCSE'00: Proceedings of the thirty-first SIGCSE Technical Symposium on Computer Science Education, 11-15. 2000.

[82] Kastens, U. GAG: A Practical Compiler Generator. Lecture Notes in Computer Science 141, Springer. 1982.

[83] Kastens, U., Sloane, A.M., Waite, W.M. Generating Software from Specifications. Jones & Bartlett Learning. 2007.

[84] Kastens, U., Waite, W.M. Modularity and Reusability in Attribute Grammars. Acta Informatica, 31(7): 601-627. 1994.

[85] Katwijk, J. A Preprocessor for YACC or a Poor Man's Approach to Parsing Attributed Grammar. ACM SIGPLAN Notices, 18(10): 12-15. 1983.

[86] Kennedy, K., Ramanathan, J. A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing. ACM Transaction of Programming Languages and Systems, 1(1): 142-160. 1979.

[87] Kiili, K. Digital Game-based Learning: Towards an Experiential Gaming Model. The Internet and Higher Education, 8(1): 13-24. 2005.

[88] Kleppe, A. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley. 2008.

[89] Knuth, D.E. Semantics of Context-free Languages, Mathematical System Theory, 2(2):127-145. 1968.

[90] Knuth, D.E. Semantics of Context-free Languages: Correction. Mathematical Systems Theory, 5(1): 95-96. 1971.

[91] Kodaganallur, V. Incorporating Language Processing into Java applications: a JavaCC Tutorial. IEEE Software, 21(4): 70-77. 2004.

[92] Koper, E.J.R., Manderveld, J.M. Educational Modelling Language: Modelling Reusable, Interoperable, Rich and Personalised Units of Learning. British Journal of Educational Technology, 35(5): 537-552. 2004.

[93] Lage, F.J., Zubenko, Y., Cataldi, A. An Extended Methodology for Educational Software Design: Some Critical Points. En FIE2001: Proceedings of the 31st Annual Conference Frontiers in Education. 1: 13-18. 2001.

[94] Larraza-Mendiluze, E., Garay-Vitoria, N. Changing the Learning Process of the Input/Output Topic Using a Game in a Portable Console. En ITiCSE'10: Proceedings of the fifteenth Annual Conference on Innovation and Technology in Computer Science Education, 316-316. 2010.

[95] Ledgard, H.F. Ten Mini-languages: A Study of Topical Issues in Programming Languages. ACM Computing Surveys, 3(3): 115-146. 1971.

[96] Leitner, S. So Lernt Man Lernen. Der WegzumErfolg. Herder, Freiburg. 2003.

[97] Levine, J. Flex & Bison: Text Processing Tools. O'Reilly Media. 2009.

[98] Levine, J., Brown, D., Mason, T. Lex & Yacc (Second Edition). O'Reilly Media. 1995.

[99] Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E. Attributed Translations. Journal of Computer and System Sciences, 9(3): 279-307. 1974.

[100] Louden, K.C. Compiler Construction: Principles and Practice. Cengage Learning Editores. 1997.

[101] Magnusson, E., Hedin, G. Circular Reference Attributed Grammars—Their Evaluation and Applications. Science of Computer Programming, 68(1): 21-37. 2007.

[102] Mallozzi, J.S.. Thoughts on and Tools for Teaching Compiler Construction. Journal of Computing in Small Colleges, 21(2):177-184. 2005.

[103] Maran, N.J., Glavin, R J. Low- to High-fidelity Simulation – a Continuum of Medical Education?. Medical Education,37(s1): 22-28. 2003.

[104] Marshall, S., Mitchell, G. Applying SPICE to e-Learning: an e-Learning Maturity Model. Proceedings of Sixth Australasian Conference on Computing Education, 30: 185-191. 2004.

[105] Martínez-Ortiz, I., Sierra, J.L., Fernández-Manjón, B., Fernández-Valmayor, A. Language Engineering Techniques for the Development of e-Learning Applications. Journal of Network and Computer Applications, 32(5): 1092-1105. 2009.

[106] McPeak, S., Necula, G.C. Elkhound: A Fast, Practical GLR Parser Generator. En CC'04: International Conference on Compiler Construction. Lecture Notes in Computer Science, 2985: 73-88. 2005.

[107] Melinchar, B. Syntax Directed Translation with LR Parsing. En CC'92: Proceedings of the 4th International Conference on Compiler Construction, 30-36. 1992.

[108] Mernik, M., Lenic, M., Acdicausevic, E., Zumer, V. LISA: An Interactive Environment for Programming Language Development. En CC'02: 11th International Conference on Compiler Construction. Lecture Notes in Computer Science, 2304: 1-4. 2002.

[109] Mernik, M., Zumer, V. An Educational Tool for Teaching Compiler Construction. IEEE Transactions on Education, 46: 61-68. 2003.

[110] Minua, M., Andreas, O., Lakhmi, J. Serious Games and Edutainment Applications. Springer Verlag. 2011.

[111] Morales, R., Leeder D., Boyle, T. A Case in the Design of Generative Learning Objects (GLO): Applied Statistical Methods GLOs. En ED-MEDIA: World Conference on Educational Multimedia, Hypermedia and Telecommunications, 2005(1): 2091-2097. 2005.

[112] Moreno-Ger, P., Martínez-Ortiz, I., Sierra, J.L., Fernández-Manjón, B. A Content-Centric Development Process Model. IEEE Computer, 41(3): 24-30. 2008.

[113] Moreno-Ger, P., Sierra, J.L., Martínez-Ortiz, I., Fernández-Manjón, B. A Documental Approach to Adventure Game Development. Science of Computer Programming, 67(1): 3-31. 2007.

[114] Mössenböck, H. A Generator for Production Quality Compilers. CC'90: 3rd International Workshop on Compiler Compilers, Lecture Notes in Computer Science, 477: 42–55. 1990.

[115] Nadolski, R.J., Hummel, H.G., Van Den Brink, H.J., Hoefakker, R.E., Slootmaker, A., Kurvers, H. J., Storm, J. EMERGO: A Methodology and

Toolkit for Developing Serious Games in Higher Education. Simulation & Gaming, 39(3): 338-352. 2008.

[116] Natvig, L., Line, S. Age of Computers: Game-based Teaching of Computer Fundamentals. En ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, 107-111. 2004.

[117] Navarro, E., Hoek, A. Multi-site Evaluation of SimSE. En SIGCSE '09: Proceedings of the 40th ACM Technical Symposium on Computer Science Education, 326-330. 2009

[118] Oliveira, N., Varanda-Pereira, M.J., Henriques, P.R., da Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science and Information Systems, 7(2): 266-289. 2010.

[119] Ong, J. C., Remolina, E., Smith, D. E., Boddy, M. S. A Visual Integrated Development Environment for Automated Planning Domain Models. En AIAA Space 2013 Conference. 2013.

[120] Paakki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computer Surveys, 27(2): 196-255. 1995.

[121] Paakki, J. PROFIT: A System Integrating Logic Programming and Attribute Grammars. En PLILP'91: 3rd International Symposium on Programming Language Implementation and Logic Programming. Lecture Notes in Computer Science, 538: 243-254. 1991.

[122] Paakki, J. Prolog in Practical Compiler Writing. Computer Journal, 34(1): 64-72. 1991.

[123] Paquette, G., Rosca, I., De la Teja, I., Léonard, M., Lundgren-Cayrol, K. Web-based Support for the Instructional Engineering of E-learning Systems. En WebNet'01 Conference Proceedings, 981-987. 2001.

[124] Parr, T. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. 2007.

[125] Parr, T., Fisher, K. LL(*): the Foundation of the ANTLR Parser Generator. ACM SIGPLAN Notices, 46(6): 425-436. 2011.

[126] Paxson, V., Estes, W., Millaway, J. Flex: the Fast Lexical Analyzer. http://www. gnu. org/software/flex. 2012.

[127] Plaisant, C. The Challenge of Information Visualization Evaluation. En AVI'04: Proceedings of the Working Conference on Advanced Visual Interfaces, 109-116. 2004.

[128] Rebernak, D., Mernik, M., Henriques, P.R., Carneiro, D., Varanda-Pereira, M.J. Specifying Languages Using Aspect-oriented Approach: AspectLISA. Journal of Computing and Information Technology, 4: 343-350. 2006.

[129] Rebernak, D., Mernik, M., Henriques, P.R., Varanda-Pereira, M.J. AspectLISA: An Aspect-oriented Compiler Construction System Based on Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, 164: 37-53. 2006.

[130] Resler, D., Deaver, D. VCOCO: A Visualisation Tool for Teaching Compilers. En ITiCSE'98: Proceedings of the 6th annual Conference on the Teaching of Computing and the 3rd annual Conference on Integrating Technology into Computer Science Education, 199-202. 1998.

[131] Robbins, S. A Java Execution Simulator. En SIGCSE '07: Pproceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, 536-540. 2007.

[132] Robbins, S. A UNIX Concurrent I/O Simulator. En SIGCSE '06 : proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, 303-307. 2006.

[133] Robbins, S. An Address Translation Simulator. En SIGCSE '05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, 515-519. 2005.

[134] Rodger, S. H., Wiebe, E., Lee, K. M., Morgan, C., Omar, K., Su, J. Increasing Engagement in Automata Theory with JFLAP. ACM SIGCSE Bulletin, 41(1): 403-407. 2009.

[135] Rodríguez-Cerezo D., Sarasa-Cabezuelo A., Gómez-Albarrán M., Sierra-Rodríguez J.L. User-Centered Development of Generative Educational Systems for Computer Engineering: The Evaluators Case Study. International Journal of Engineering Education, 31(3): 751–763. 2015.

[136] Rodríguez-Cerezo D., Sierra, J.L. Introducing a Design-Preserving Implementation Strategy in a Compiler Construction Course. En SIIE'13: actas del XV Simposio Internacional de Informática Educativa, 24-29. 2013.

[137] Rodriguez-Cerezo, D., Gomez-Albarrán, M., Sierra, J. L. From Collections of Exercises to Educational Games: A Process Model and a Case Study. En ICALT'11: Proceedings of the 11th IEEE International Conference on Advanced Learning Technologies, 282-284. 2011.

[138] Rodríguez-Cerezo, D., Gómez-Albarrán, M., Sierra, J.L. A Process Model for the Generative Production of Interactive Simulations in Engineering Education. En TEEM'13: proceedings of the First International Conference on Technological Ecosystem for Enhancing Multiculturality, 95-103. 2013.

[139] Rodriguez-Cerezo, D., Gómez-Albarrán, M., Sierra-Rodríguez, J.L. Interactive Educational Simulations for Promoting the Comprehension of Basic Compiler Construction Concepts. En ITiCSE'13: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education, 28-33. 2013.

[140] Rodriguez-Cerezo, D., Henriques, P. R., Sierra, J. L. Attribute Grammars Made Easier: EvDebugger a Visual Debugger for Attribute Grammars. En SIIE'14: Proceedings of International Symposium on Computers in Education, 23-28. 2014.

[141] Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J.L. Serious Games in Tertiary Education: A Case Study Concerning the Comprehension of Basic Concepts in Computer Language

Implementation Courses. Computers in Human Behavior, 31: 558-570. 2014.

[142] Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Gómez-Albarrán, M., Sierra, J.L. Facilitating Comprehension of Basic Concepts in Computer Language Implementation Courses: A Game-based Approach. En SIIE'12: International Symposium on Computers in Education , 1-6. 2012.

[143] Rodríguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J. L. A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools. Computer Science and Information Systems, 9(3): 983-1017. 2012.

[144] Rodriguez-Cerezo, D., Sarasa-Cabezuelo, A., Sierra, J.L. Implementing Attribute Grammars Using Conventional Compiler Construction Tools. En FedCSIS'11: proceedings of Federated Conference on Computer Science, 855-862. 2011.

[145] Ruckert, M. Teaching Compiler Construction and Language Design: Making the Case for Unusual Compiler Projects with Postscript as the Target Language. En SIGCSE'07: Proceedings of the twenty-ninth SIGCSE Technical Symposium on Computer Science Education, 232-236.1998.

[146] Sadler, D.R. Formative Assessment and the Development of Instructional Systems. Instructional Science, 18(12): 119-144. 1989.

[147] Santos, P.A., Castro-Sánchez, J.J. Una Herramienta para la Enseñanza y Aprendizaje de la Asignatura Procesadores de Lenguajes. En JENUI'06: Actas de las XII Jornadas de la Enseñanza Universitaria de la Informática, 499-507. 2006.

[148] Saraiva, J., Swiestra, D. Generic Attribute Grammars. En WAGA'99: Proceedings of the 2nd Workshop on Attribute Grammars and Their Applications. 1999.

[149] Sarasa, A., Temprado-Battad, B., Sierra, J.L, Fernández-Valmayor, A. XML Language-Oriented Processing with XLOP. En WAINA'09: proceedings of 5th International Symposium on Web and Mobile Information Services, 322-327. 2009.

[150] Sarasa-Cabezuelo, A., Sierra, J.L. The grammatical approach: A Syntax-directed Declarative Specification Method for XML Processing Tasks. Computer Standards & Interfaces, 35(1): 114-131. 2013.

[151] Sarasa-Cabezuelo, A., Sierra-Rodríguez, J.L. Software Engineering for eLearning. En TEEM'13: Proceedings of the First International Conference on Technological Ecosystem for Enhancing Multiculturality, 81-86. 2013.

[152] Sassa, M., Ishizuka, H., Nakata, I. Rie, a Compiler Generator Based on a One Pass-type Attribute Grammar. Software – Practice & Experience, 25(3): 229-250. 1995.

[153] Sathi, H.L. A Project-based Course in Compiler Construction. En SIGCSE'86: Proceedings of the seventeenth SIGCSE Technical Symposium on Computer Science Education, 114-119. 1986.

[154] Schreiner, A.T., Friedman, H.G. Introduction to Compiler Construction with Unix. Prentice-Hall. 1985.

[155] Scott, E., Johnstone, A. Right Nulled GLR Parsers. ACM Transactions on Programming Languages and Systems, 28(4): 577-618. 2006.

[156] Shaffer, D.W., Squire, K.R., Richard, H., Gee, J.P. Video Games and the Future of Learning. Phi Delta Kappan, 87(2): 104-111. 2005.

[157] Shapiro, H.D., Mickunas, M. D. A New Approach to Teaching a First Course in Compiler Construction. En SIGCSE' 76: Proceedings of the ACM SIGCSE- SIGCUE Technical Symposium on Computer Science and Education, 158-166. 1976.

[158] Sierra, J.L., Fernández-Pampillón, A.M., Fernández-Valmayor, A. An Environment for Supporting Active Learning in Courses on Language Processing. En ITiCSE'08: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, 128-132. 2008.

[159] Sierra, J.L., Fernández-Valmayor, A. A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars. Electronic Notes in Theoretical Computer Science, 19-36. 2006.

[160] Sterling, L., Shapiro, E. The Art of Prolog, 2nd Edition. 1994.

[161] Stevens, B.,L., Frank L. L. Aircraft Control and Simulation. John Wiley & Sons. 2003.

[162] Štuikys, V., Damaševicius, R. Development of Generative Learning Objects Using Feature Diagrams and Generative Techniques. Informatics in Education, 7(2): 277–288. 2008.

[163] Štuikys, V., Damaševicius, R. Towards Knowledge-based Generative Learning Objects. Information Technology and Control, 36(2): 202-212. 2007.

[164] Temprado-Battad, B., Sarasa, A., Sierra, J.L. Using Attribute Grammars to Manage the Production and Evolution of e-Learning Tools. En ICALT'10: Proceedings of the 9th International Conference on Advanced Learning Technologies, 427-431. 2010.

[165] Urquiza-Fuentes, J., Gallego-Carrillo, M., Gortázar-Bellas, F., Velázquez-Iturbide, J.Á. Visualizing the Symbol Table. En ITiCSE'06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, 341-341. 2006.

[166] Urquiza-Fuentes, J., Velázquez-Iturbide, J.Á. A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems. ACM Transactions on Computing Education, 9(2): 24pp. 2009.

[167] Varanda-Pereira, M. J., Oliveira, N., Cruz, D., Henriques, P. Choosing Grammars to Support Language Processing Courses. En SLATE'13: Proceedings of the 2nd Symposium on Languages, Applications and Technologies, 155-168. 2013.

[168] Vegdahl, S.R. Using Visualization Tools to Teach Compiler Design. En Journal of Computing Sciences in Colleges, 16(2): 72-83. 2000.

[169] Velázquez-Iturbide, J.A. Errores, Actitudes y Malentendidos en la Experimentación con Algoritmos de Optimización. En SIIE'14: Actas del XVI Simposio Internacional de Informática Educativa, 37-43. 2014.

[170] Vidani, A. C., Chittaro, L. Using a Task Modeling Formalism in the Design of Serious Games for Emergency Medical Procedures. En VS-GAMES'09: Proceedings of Games and Virtual Worlds for Serious Applications, 95-102. 2009.

[171] Voelter,M. DSL Engineering: Designing, Implementing and Using Domain-Specific Languages. CreateSpace Independent Publishing Platform. 2013.

[172] Vogt, H.H., Swierstra, S.D., Kuiper, M.F. Higher-Order Attribute Grammars. ACM SIGPLAN Notices, 24(7): 131-145. 1989.

[173] Vyk, E.R., Schwerdfeger, A.C. Context-aware Scanning for Parsing Extensible Languages. En GPCE'06: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, 63-72. 2006.

[174] Waite, W.M., Jarrahian, A., Jackson, M.H., Diwan, A. Design and Implementation of a Modern Compiler Course. En ITiCSE'06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, 18-22. 2006.

[175] Walsteijn, M.J., Kuiper, M.F. Attribute Grammars in Prolog. Technical Report, RU-CS-86-14. 1986.

[176] Ware, C. Information Visualization Perception for Design (2nd Edition). Elsevier. 2006.

[177] Werner, M. A Parser Project in a Programming Languages Course. Journal of Computing in Small Colleges, 18(5):184-192. 2003.

[178] White, E., Sen, R., Stewart, N. Hide and show: using real compiler code for teaching. En SIGCSE'05: Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, 12-16. 2005.

[179] White, T.M., Way, T.P. jFAST: a Java Finite Automata Simulator. En SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, 384-388. 2006.

[180] Woolf, B.P. Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing e-Learning. Morgan-Kaufman. 2008.

[181] Wyk, E.V., Bodin, D., Gao, J., Krishnan, L. Silver: An Extensible Attribute Grammar System. Science of Computer Programming, 75(1-2): 39-54. 2010.

[182] Xu, L., Martin, F.G. Chirp on Crickets: Teaching Compilers Using an Embedded Robot Controller. En SIGCSE' 06: Proceedings of the 37th SIGSE Technical Symposium on Computer Science Education, 82-86. 2006.