ACIDE

ACIDE

TRABAJO FIN DE GRADO

CURSO 2022-2023

AUTOR

CARLOS SEGUIDO MARTÍNEZ

DIRECTOR

PROF. FERNANDO SÁENZ PÉREZ

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

ACIDE

ACIDE

TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA

AUTOR

CARLOS SEGUIDO MARTÍNEZ

DIRECTOR

PROF. FERNANDO SÁENZ PÉREZ

CONVOCATORIA: FEBRERO 2023

GRADO EN INGENIERÍA INFORMÁTICA

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

9 DE ENERO DE 2023

Resumen

ACIDE (A Configurable IDE) es un entorno gráfico altamente configurable de código abierto, gratuito y multiplataforma que puede ser usado para desarrollar intérpretes, bases de datos, etc.

La tarea a completar consiste en la creación de un analizador que permita la obtención de una descripción del lenguaje del código creada por el usuario y permita el coloreado de elementos léxicos y la comprobación de la corrección sintáctica.

Palabras Clave
ACIDE, analizador, léxico, sintáctico, AFD, LALR(1), construcción de subconjuntos

Abstract

ACIDE (A Configurable IDE) is a highly configurable, open source, multiplatform and free enviroment that can be used to develop parsers, databases, etc.

The task to complete consists in the development of an analyzer that allows the user to provide a custom description of a programming language which is used for the highlighting of lexical elements and the syntactic correction check.

Keywords
ACIDE, analyzer, lexical, syntactic, DFA, LALR(1), subset construction

Índice

Capítulo 1 - Introducción

En esta memoria se realiza una descripción del diseño e implementación de un sistema que permita al usuario definir su propia gramática que luego podrá ser empleada por ACIDE para resaltar el código producido en su editor de texto y comprobar la corrección de la sintaxis al escribir programas en ese lenguaje. Para ello se hará uso de un analizador léxico, uno sintáctico y otro semántico

En los siguientes apartados se explicará de manera detallada la implementación de las distintas partes del compilador.

1.1 Motivación

Durante el transcurso de mi grado en Ingeniería Informática en la Complutense he adquirido un gusto por la algoritmia y las estructuras de datos, el cual me ha llevado a escoger ACIDE como un proyecto para el trabajo de fin de grado pues considero que la tarea a tratar en cuestión se acerca considerablemente a los campos ya mencionados.

El cometido de este proyecto trata de otorgar a ACIDE la capacidad de analizar lenguajes descritos por el propio usuario e identificar errores producidos en la producción de dicho código.

ACIDE nace de la necesidad de entornos que permitan la depuración de bases de datos SQL (Structured Query Language). Este entorno ofrece una interfaz gráfica que junto a DES (Datalog Educational System), que permite depurar el código textualmente, permite cubrir esta necesidad. Con la finalidad de ofrecer más funcionalidades, el desarrollo de ACIDE ha sido continuado a lo largo de varios años en distintos trabajos de fin de grado.

1.2 Objetivos

El objetivo del trabajo de fin de grado consiste en la implementación e incorporación de las funcionalidades pertenecientes a un compilador capaces de destacar en diversos colores y reconocer:

- Los distintos *tokens de un lenguaje especificado por el usuario, es decir, un analizador léxico.*
- *La sintaxis del lenguaje y su corrección, es decir, un analizador sintáctico.*
- Opcionalmente, identificar los tipos y verificar su correcto uso, es decir, un analizador semántico.

Adicionalmente el análisis debe realizarse de forma progresiva, en tiempo real, durante la escritura del código del programa. Para ello debe permitirse, además de la escritura, el reemplazamiento de código, la eliminación de código y la escritura "in medias res", es decir, en cualquier lugar entre el inicio y el final.

1.2.1 Objetivos no alcanzados

Se ha logrado la realización de los analizadores léxico y sintáctico y, parcialmente, del semántico, además del destacado del código en diversos colores especificados por el usuario.

Se ha logrado el análisis progresivo en tiempo real. No se ha logrado el reemplazamiento de código, la eliminación ni la escritura en medio del programa.

1.3 Antecedentes del trabajo

ACIDE es un IDE (*integrated development enviroment*) que puede ser usado como intérprete, compilador o sistemas de bases de datos, como SQL entre otros. Está constituido como un proyecto en constante evolución debido al desarrollo de la herramienta por distintos grupos a lo largo de los años, siempre bajo la dirección de Fernando Sáenz Pérez.

- Durante el curso 2006-2007, por Diego Cardiel Freire, Juan José Ortiz Sánchez y Delfín Rupérez Cañas.
- Durante el curso 2007-2008, por Miguel Martín Lázaro.
- Durante el curso 2010-2011, por Javier Salcedo Gómez.
- Durante el curso 2012-2013, por Pablo Gutiérrez García-Pardo, Elena Tejeiro Pérez de Ágreda y Andrés Vicente del Cura.
- Durante el curso 2013-2014, por Juan Jesús Marqués Ortiz, Fernando Ordás Lorente y Semíramis Gutiérrez Quintana.
- Durante el curso 2014-2015, por Sergio Domínguez Fuentes.
- Durante el curso 2019-2020, por Sergio García Rodríguez.
- Durante el curso 2020-2021, por Carlos González Torres.

Debido al tipo de desarrollo se han establecido ciertos estándares que han de seguirse en el desarrollo de código para ACIDE.

1.3.1 Estándares de código fuente

Como se ha mencionado antes, ACIDE se trata de un proyecto desarrollado por varios desarrolladores a lo largo de una cantidad considerable de tiempo. Para facilitar la mantenibilidad del código y la futura implementación de nuevas funcionalidades, se ha visto

necesario establecer unos estándares que deben seguir los desarrolladores;

- Todos los comentarios e identificadores deben ser declarados en inglés.
- Al inicio de cualquier clase del proyecto debe indicarse su licencia: GPLv3.

```
/*
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
```

Figura 1-1 Licencia GPLv3

- Introducción de comentarios en cada función y atributo del código para indicar su propósito.
- Cada clase de Java contará con un comentario en Javadoc con el siguiente formato:

```
/**
 *      Descripción de la clase
 *
 *      @version version
 * @see <NombreDeClase/NombreDeInterfaz>
 */
```

Figura 1-2 Comentario en Javadoc de las clases

- Todos los atributos de clase empezarán por "_". Por ejemplo: "_transitionTable"
- La denominación de las clases empieza por "Acide" seguido de su nombre. Por ejemplo: "AcideLexAnalyzer".
- Todas las palabras de los nombres empezarán por mayúscula tanto en atributos: "_transitionTable", clases "AcideLexAnalyzer" o métodos: "applyEpsilonClosure()".
- Las variables constantes tendrán su nombre completo en mayúscula y si está formado por varias palabras, estas deberán separarse por "_". Por ejemplo: "EMPTY_TRANSITION".
- Todas las clases que crean ventanas de la aplicación deberán incluir los siguientes métodos:

```
// Builds the ACIDE - A Configurable IDE configuration window components
private void buildComponents() {}

//Adds the components to the ACIDE - A Configurable IDE to the
//configuration window
private void addComponents() {}

//Sets the text of the ACIDE - A Configurable IDE class components
//with the labels in the selected language to display
private void setTextOfMenuComponents() {}

//Updates the ACIDE - A Configurable IDE class components
//visiblilty with the menu configuration
private void updateComponentsVisibility() {}

//Sets the listeners of the configuration window components
private void setListeners() {}
```

Figura 1-3 Métodos obligatorios para nuevas ventanas

1.4 Plan de trabajo

Al inicio del curso el profesor Fernando Sáenz Pérez, director del proyecto, me facilitó el acceso a memorias del proyecto de años anteriores, así como su manual y el código fuente.

Con este material (y a falta de cursar la asignatura de PL) se tomó el inicio del proyecto mediante una identificación de los requisitos, que culminaría en los objetivos anteriormente descritos.

Posteriormente se realizó una lluvia de ideas sobre cómo abordar las distintas necesidades a satisfacer por el proyecto entre manos.

Más adelante, (durante el curso de PL) se estableció una metodología más rigurosa sobre la resolución del proyecto, consistente en identificar un problema, si es sencillo, resolverlo o en caso contrario dividirlo en problemas más sencillos. Repetir este proceso hasta completar el diseño del proyecto.

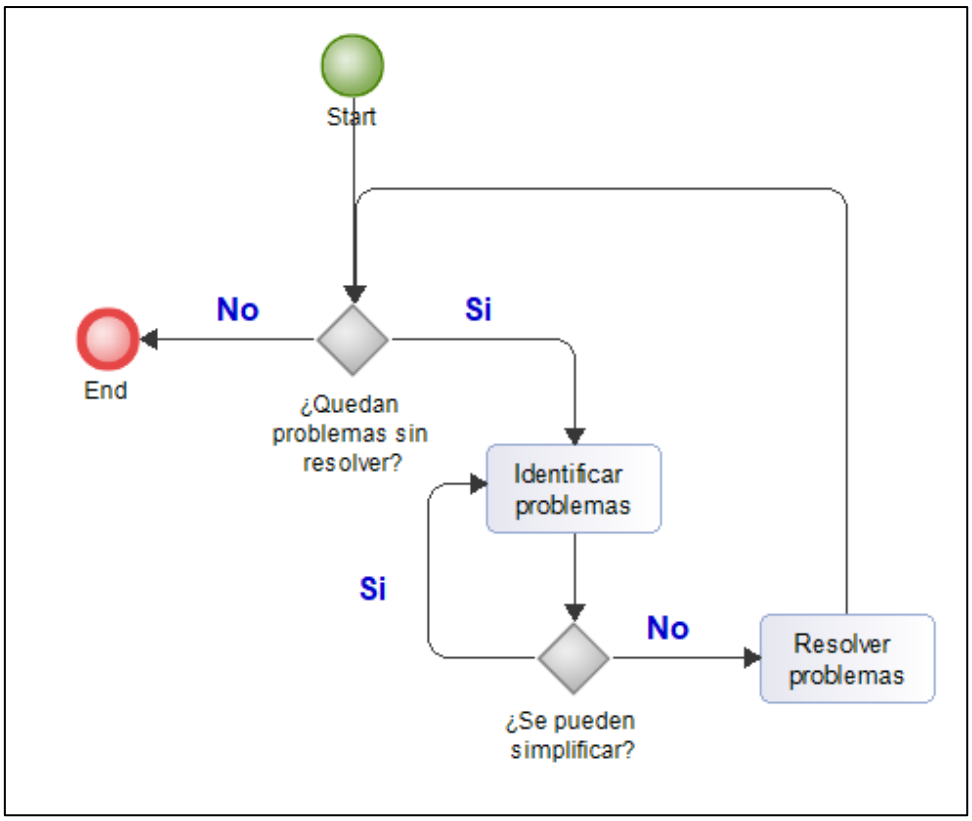

Figura 1-4 Diagrama de flujo de plan de trabajo

Una vez terminado el desarrollo del diseño se procedió a implementarlo en código, depurarlo y en caso de encontrar fallos en el diseño, corregirlos mediante el método ya explicado.

Una vez terminado el analizador, se procedió a su incorporación al sistema de acide. Dicha incorporación se realizó mediante la identificación de código a modificar y añadir ventanas donde se identificase como necesario de tal forma que se dejase el código original de ACIDE con una cantidad de modificaciones mínimas al tiempo que se agregaba una funcionalidad completamente nueva.

1.5 Cronología

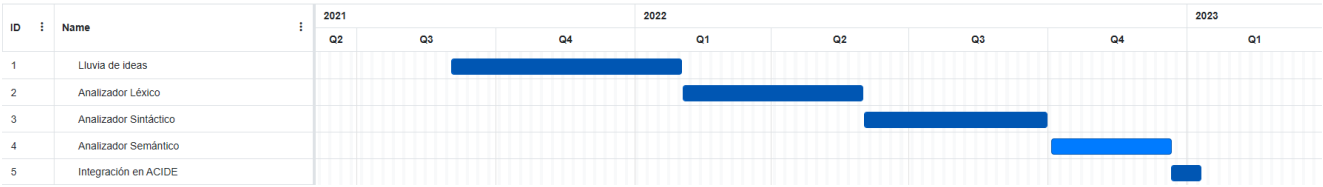

Figura 1-5 Diagrama de la cronología de tareas

Capítulo 2 - Diseño

2.1 Analizador léxico

El analizador léxico, una clase que representa un autómata finito determinista envuelta en otra clase que interpreta el estado actual y devuelve el *token* pertinente.

Se divide en tres partes: representación de los estados, transición entre estados, analizador que recibe un texto y lo convierte en un analizador léxico.

2.1.1 Representación de estados

El autómata finito determinista contiene cuatro atributos, cinco con la clase que interpreta el estado y devuelve el *token*, además de cualquier variable constante declarada:

- Un entero que representa el estado inicial.
- Un Set de enteros que representa los múltiples estados finales.
- Un entero que representa el estado actual.
- Una lista de mapas de carácter-enteros que representa la tabla de transiciones donde los índices de la lista representan los estados, los caracteres que funcionan como las claves del mapa representan el elemento leído y los valores del mapa representan el siguiente estado actual. Es decir:

  List<Map<Character, Integer>>
- Un mapa entero-string que recibe un estado y devuelve el *token* que representa si es un estado final.

2.1.2 Transición entre estados

Aquí hay que hacer distinción entre dos posibles tipos de transiciones: Si es posible moverse desde el estado actual o si no lo es.

Si es posible moverse, se actualiza el estado actual al nuevo estado y se comprueba si este es final. Si es final se devuelve el *token* que identifica, si no lo es se devuelve una cadena vacía para indicar que se sigue reconociendo.

Si no se puede mover, se devuelve *null*, indicando que no hay transiciones posibles y se resetea el autómata a su estado inicial.

2.1.3 *Parser*

La función del *parser* se encarga de leer un texto y convertirlo en el autómata. Para ello se distinguen dos funciones: Crear un autómata que reconozca solo un tipo de *token* y unir dos autómatas que reconocen distintos *tokens* en uno solo que reconozca ambos tipos.

Antes de interpretar cualquier texto para generar un autómata, hay que especificar qué lee el *parser* y cómo lo interpreta.

El *parser* lee una serie de líneas donde cada línea representa un único *token* y su definición. Por ejemplo:

Int_identifier := int
vowel_starting_var := (a|e|i|o|u)var
whitespace := \s

De estos ejemplos podemos extraer que cada *token* está separado de su definición por ":=" quedando cada línea de la forma

"*token* := definición" además permite caracteres especiales (\s, \n, \\, ...) y los puede interpretar.

El *parser* permite el uso de las operaciones de unión, representada por " | ", cierre en estrella o cierre de Kleene, representado por "*", cierre positivo, representado por "+" y concatenación, cualesquiera dos símbolos que se encuentran adyacentes.

Mientras que el *parser* no permite la sustitución en las definiciones por otras de carácter auxiliar, esto se solventa reemplazando las menciones a *tokens* en las definiciones por las definiciones de esos mismos *tokens*, rodeadas por paréntesis, antes de crear ningún autómata.

Una vez especificado qué lee el *parser*, ahora podemos explicar cómo funciona. El *parser* funciona de una forma simple, crea un árbol donde los nodos hoja son caracteres de entrada en el autómata y las raíces son las operaciones. El árbol está organizado de forma que las operaciones con mayor prioridad estén más cerca de las hojas que de la raíz. Este árbol es creado siguiendo de forma recursiva las siguientes opciones:

- Si la cadena tiene un solo elemento. Crear nodo hoja con ese elemento.
- Si la cadena tiene dos elementos y el primer elemento es "\". Crear un nodo hoja con un carácter especial dependiendo del segundo elemento.
- Si encontramos una operación de unión que no esté dentro de un paréntesis. Crear un árbol cuya raíz representa la operación de unión y las hojas los árboles creados por cada operando.

- Si no hay operación de unión y encontramos una operación de concatenación que no esté dentro de un paréntesis. Crear un árbol con raíz concatenación y hojas los árboles creados por cada operando.
- Si no hay ni uniones ni concatenaciones. Si el último carácter de la cadena es "+" realizamos un cierre positivo sobre el árbol formado por el resto de la cadena, si es "*" realizamos un cierre estrella.
- Si el primer carácter es "(" y el último es ")" devolvemos el árbol generado por la cadena sin estos paréntesis.
- Si se alcanza este paso devolver un error, la cadena está mal formada.

Una vez formado el árbol vamos generando eNFAs (empty transitions Non-deterministic Finite Automatas) recorriendo el árbol en postorden y creando los siguientes autómatas según la operación:

- Si es una unión:

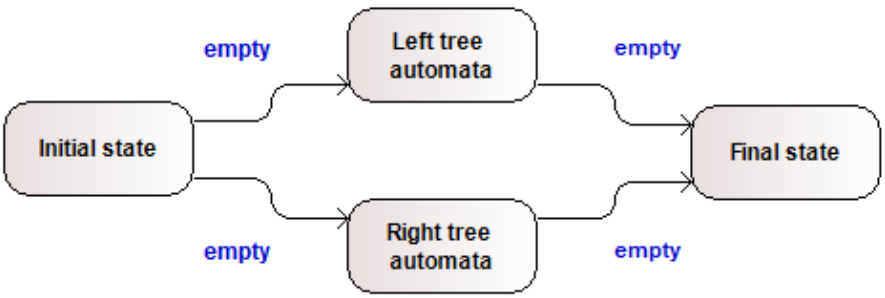

Figura 2-1 Operación de unión de dos autómatas

- Si es una concatenación:

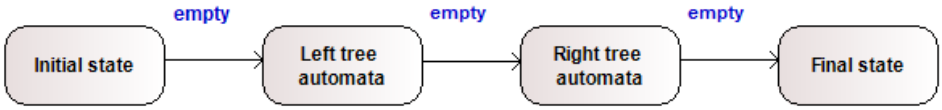

Figura 2-2 Operación de concatenación de dos autómatas

- Si es un cierre positivo:

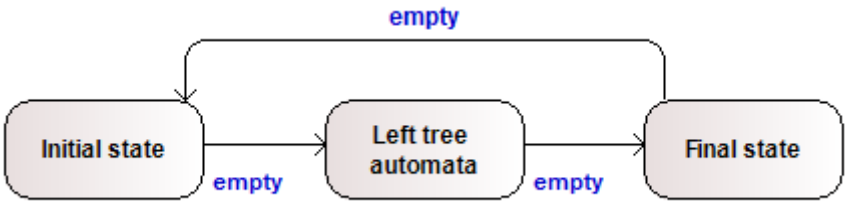

Figura 2-3 Operación de cierre positivo sobre un autómata

- Si es un cierre en estrella:

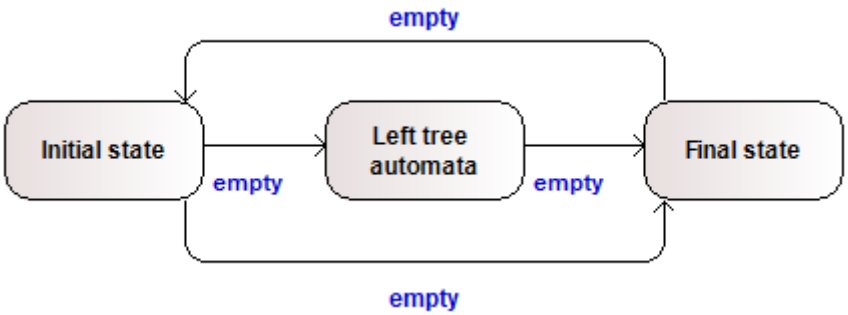

Figura 2-4 Operación de cierre de Kleene sobre un autómata

- Si reconoce un único carácter, es decir, es un nodo hoja:

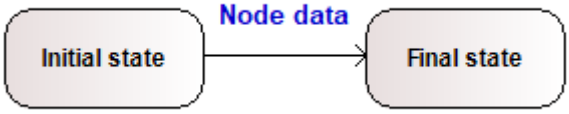

Figura 2-5 Creación de un autómata que reconoce un carácter

Una vez se ha terminado de recorrer el árbol tenemos un eNFA que reconoce un *token*. Al ser no determinista su uso durante la ejecución del programa puede volverse pesada, por tanto, el siguiente paso es convertirlo en un DFA (Deterministic Finite Automata) mediante el algoritmo de construcción de subconjuntos[1]. Una vez tenemos el DFA podemos unirlo al analizador léxico. Para unirlo se siguen los siguientes pasos:

- Guardamos la longitud actual de la lista que representa la tabla de transición del analizador léxico, la representaremos con "n".
- Añadimos a la lista del analizador los objetos de la tabla de transición del DFA sumándoles a las transiciones la longitud anteriormente guardada, es decir:

$$\delta(i, a) = j \rightarrow \delta(i + n, a) = j + n$$

- Añadimos una transición vacía desde el estado inicial al estado n:

$$\delta(0, \epsilon) = n$$

- Guardamos los nuevos estados finales del DFA y su *token*, hacemos lo mismo con todos los estados del analizador.
- Aplicamos el algoritmo de construcción de subconjuntos
- A cada nuevo estado final, si contiene uno de los anteriores estados finales, le asignamos ese mismo *token*.
- Si hay conflicto al reconocer dos *tokens* en un estado final, podemos hacer dos cosas: establecer una prioridad; si usamos el *token* del DFA entonces las definiciones más bajas en la lista pueden sobrescribir a las anteriores, en caso contrario no podrán sobrescribir. O devolver un error. En este caso se ha escogido sobrescribir.

2.1.3.1 Algoritmo de construcción de subconjuntos

El algoritmo de construcción de subconjuntos requiere una operación: $\epsilon Closure(q)$, el conjunto de estados alcanzables desde el estado q usando transiciones vacías incluido el mismo.

Los estados del nuevo autómata resultado del algoritmo estarán constituidos de conjuntos de estados del autómata original.

El algoritmo empieza generando el estado inicial a partir de $\epsilon Closure(q_0)$ siendo q_0 el estado inicial. A continuación, generamos los siguientes estados aplicando transiciones al nuevo estado inicial, a estos nuevos estados también se les aplica $\epsilon Closure$. Si alguno de estos estados ya existe, se crea una nueva transición a ese estado, si no existe, se agrega al nuevo autómata y se añade la transición.

2.2 Analizador sintáctico

El analizador sintáctico es un autómata que reconoce lenguajes LALR(1)[2] (Look Ahead LR(1)) y, al igual que el analizador léxico, está compuesto de tres partes: representación de los estados, transición entre estados y el analizador que transforma una definición en texto en el propio autómata.

2.2.1 Representación de estados

El analizador sintáctico consiste en un autómata con pila con seis atributos, nueve si contamos los añadidos por el analizador semántico:

- Una tabla de transiciones igual que la del analizador léxico cuya única diferencia es que en lugar de devolver un entero devuelve dos. El primer entero indica si la operación en un shift/goto o una reducción. Según el valor del primer entero el segundo entero representará una cosa distinta: Si la operación es shift/goto entonces el segundo entero representa el estado al que hay que moverse, si la operación es reducción entonces representa la cantidad de estados que hay que retroceder.
- Una tabla de reducciones a la que se le introduce el estado actual y un *token y devuelve el símbolo no terminal que se reduce. Esta tabla se usa solo en las operaciones de reducción.*
- *Un entero que representa el estado inicial.*
- *Un entero que representa el estado de aceptación.*
- *Un entero que representa el estado actual.*
- Un stack de enteros en el que se almacena los estados por los que ha ido pasando el autómata.

2.2.2 Transición entre estados

Hay que distinguir tres tipos de transiciones: shift/goto, reducción y transiciones inválidas.

- Si la transición es invalida, es decir, no existe una transición desde el estado actual con el símbolo introducido en la tabla, en este caso no se cambia de estado y se devuelve un error.
- Si la transición es shift/goto se cambia de estado según se especifica en la tabla de transiciones. Además, se introduce en el stack el anterior estado actual.
- Si la transición es una reducción hay que realizar varias acciones:
    1. Obtenemos el nuevo símbolo no terminal de la tabla de reducciones.

2. Retrocedemos la cantidad de estados especificada en el segundo entero obtenido en la tabla de transiciones sacando esa cantidad de estados del stack.
3. Actualizamos el estado actual con el no terminal obtenido en el paso 1.
4. Volvemos a analizar el símbolo que se ha introducido en la máquina.

2.2.3 *Parser*

La función del *parser* se encarga de leer un texto que representa un autómata LALR(1). El autómata lee una opción de producción por línea, es decir:

S := A A
A := a A
A := b

Donde "a" y "b" son terminales y "S" y "A" son no terminales. El símbolo no terminal se separa de la producción con el símbolo ":=". Los no terminales y terminales de la producción se separan por espacios.

Para crear el autómata necesitamos objetos LR1 formados por un núcleo equivalente a una producción del autómata con un punto que indica la cantidad de la producción consumida y un anticipo que es un conjunto de los símbolos que pueden aparecer inmediatamente después de la producción y dos operaciones: closure y succ.

- Objeto LR1: $S' \rightarrow \cdot S$, \$ donde $S' \rightarrow \cdot S$ es el núcleo y \$ el anticipo.

- Para un conjunto de objetos LR1 q $closure(q)$ devuelve otro conjunto de objetos LR1 p tal que para cada objeto k de q si el símbolo posterior al punto es no terminal, se añade el objeto LR1 formado por ese símbolo a p. Por ejemplo:

$$S \rightarrow \cdot AA \qquad closure(S) = A \rightarrow \cdot aA$$
$$A \rightarrow \cdot b$$

- Para un conjunto de objetos LR1 q y un símbolo "a" $succ(q, a)$ devuelve un conjunto de objetos LR1 p tal que para objeto k de q si el símbolo posterior al punto es igual a "a" entonces se avanza el punto de ese objeto LR1 y añade a p.

$$A \rightarrow \cdot aA \qquad \text{SUCC(A, a)} = A \rightarrow a \cdot A$$
$$A \rightarrow \cdot b$$

El parser empieza formando el lenguaje extendido, es decir, si S es la transición inicial, agrega la transición:

S' := S

Primero producimos el objeto LR1 de la producción inicial, que ahora es S'. A este objeto se le añade un índice para poder localizarlo, por ejemplo, el objeto LR1 número 1 del estado 0: $\#_{0,1}$

$$0, S' \rightarrow \cdot S, \$$$

A este objeto se le aplica closure y obtenemos:

$$0, S' \rightarrow \cdot S, \{\$\}$$
$$1, S \rightarrow \cdot AA, \ \{\#_{0,0}\}$$
$$2, A \rightarrow \cdot aA, \ \{a, b\}$$
$$3, A \rightarrow \cdot b, \ \{a, b\}$$

Para calcular los anticipos, en el caso de closure usamos los primeros símbolos del resto del núcleo posterior al símbolo que se ha expandido, si el resto de símbolos pueden ser vacíos entonces ponemos una referencia a ese objeto.

En el caso de succ se añade directamente la referencia.

Ahora por cada símbolo tanto terminal como no terminal a se aplica a cada estado q closure(succ(q, a)) obteniendo un nuevo estado formado por un conjunto de objetos LR1. Si el estado no se había generado aun (sin tener en cuenta los anticipos) se añade a la tabla y la transición del estado original al nuevo estado se forma usando el símbolo que se usó en la función succ. Si ya existía se añade la transición simplemente Todos los objetos que tengan el punto al final serán reducciones y se guardará en la tabla reducciones. Por ejemplo, sea Δ la tabla de reducciones, S un objeto LR1 que se puede reducir y n el estado actual:

$$S \rightarrow ab. \, , \{c, d\} \Rightarrow \Delta(n, c) = S$$
$$\Delta(n, d) = S$$

Una vez producidos todos los estados se resuelven las referencias que crean un sistema de ecuaciones donde cada referencia equivale a los anticipos del objeto LR1 al que indica.

Al final tenemos un DFA cuyas transiciones son las operaciones shift/goto.

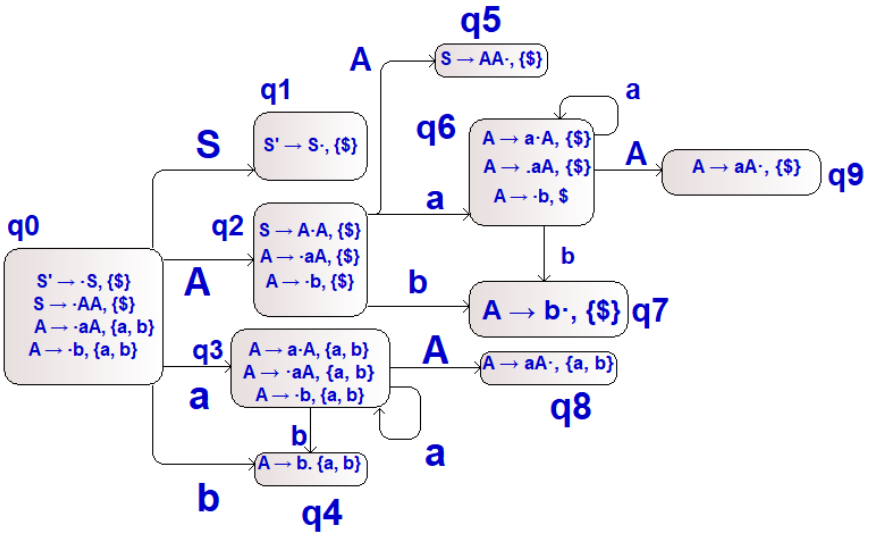

Figura 2-6 Diagrama de estados LALR(1) (contiene estados repetidos)

Para terminar de generar el analizador sintáctico, inicializamos los estados finales, el estado inicial y el estado actual. También inicializamos la pila de forma que esté vacía.

2.3 Analizador Semántico

El analizador semántico produce árboles que realizan operaciones aritmético-lógicas sobre los valores, las palabras que dan lugar a los *tokens* en el analizador léxico, cuya completitud indican la corrección de propiedades definidas por el usuario, por ejemplo, una línea del programa debe tener más margen que otra, o dos operandos de una operación deben ser el mismo tipo. El código del analizador semántico se encuentra dentro de la clase del analizador sintáctico y realiza los cálculos de los árboles en las operaciones de reducción de este último. Entre las operaciones de los árboles se incluyen el poder guardar y obtener valores de tablas en la memoria. Dividiremos este analizador en representación, cálculo y su parser.

2.3.1 Representación

Como se ha mencionado antes se trata de árboles. Por lo tanto, tenemos una raíz que indica que operación realizar y tenemos hojas, cuya cantidad puede variar de una a tres dependiendo de la operación, estas hojas pueden contener valores estáticos, más operaciones o valores dinámicos extraídos del código.

Estos árboles se guardan en tablas en el analizador sintáctico, y dependiendo de que producción se reduzca se ejecutarán unos u otros. A su vez, están categorizados en dos tipos distintos:

- Operaciones: Cuyo valor final es ignorado y solo nos interesa si se completan o no, es decir si devuelven un error entonces hay un fallo en el código del programa. Puede haber varios por opción por producción.
- Retornantes: Cuyo valor final es guardado pues se usa como valor con tendrá un símbolo no terminal en la producción que lo haya llamado. Si devuelven error también indican un fallo en el programa. Solo hay una por opción, por producción.

El analizador semántico también tiene un stack donde se guardan los valores al mismo tiempo que se consumen símbolos en la operación de shift/goto.

2.3.2 Cálculo

Cuando se realiza una operación de reducción, a la vez que se obtienen estados del stack del analizador sintáctico, se consiguen valores del stack del analizador semántico, de forma que el valor más reciente se relacione con el símbolo más posterior en una producción.

Una vez tengamos un valor por símbolo de la producción, estos se pasan a la raíz de cada árbol de operaciones, hay que recordar que puede haber varios de operaciones y uno de retorno.

Una vez entramos en el árbol, este se recorre en postorden. Ejecutando primero el subárbol de más izquierda, terminando en el de más a la derecha y por último ejecuta la operación de la propia raíz. Hay casos especiales en los que no se ejecuta el código de todas sus ramas (caso if-then-else) o casos en los que solo hay una hoja (operación not o lanzar un error).

Una vez terminado el cálculo del árbol puede ocurrir uno de los siguientes casos:

- Si el árbol devuelve un error, la operación de reducción devuelve un error semántico, la operación de reducción se completa de todas formas.
- Si el árbol se encuentra en la lista de operaciones y no devuelve error, se ignora su resultado y se prosigue con el siguiente árbol si queda.
- Si el árbol es el de retorno, su resultado se almacena en el stack de valores como valor que utilizará el símbolo no terminal en la ejecución de árboles de la producción que lo ha consumido.

2.3.3 Parser

La descripción del árbol de operaciones se encuentra agregada la descripción de la producción que lo ejecuta en su reducción de la siguiente forma:

P := a b c; operations := add(1,2), mul(2,[0]); return := sub(2,1)

Como se puede observar, en este caso habría una operación de suma y otra de multiplicación; en el caso de la suma se trata de 1+3 y en el caso de la multiplicación se trata de 2*(el valor del *token* 0, es decir, el *token* a) mientras que el valor de retorno de la producción P es la resta 2-1.

Tanto la sección de operaciones como la de retorno se pueden omitir. En el caso de las operaciones, se quedará vacía y no se ejecutará nada. En el caso del retorno se devolverá por defecto el valor "1".

A la hora de construir el árbol, este recibirá un string que represente una operación. Este string será analizado de forma que tendremos la operación a realizar y sus operandos.

Al generarse el árbol en postorden primero se generan las ramas de los operandos, de izquierda a derecha usando el mismo constructor, de forma recursiva, y pasando el string del operando. Si el operando es un valor estático, este se almacena, si es dinámico, se almacena la posición del símbolo que lo genera. Una vez generadas las operaciones que devuelven los operandos se procede a generar la operación de la raíz.

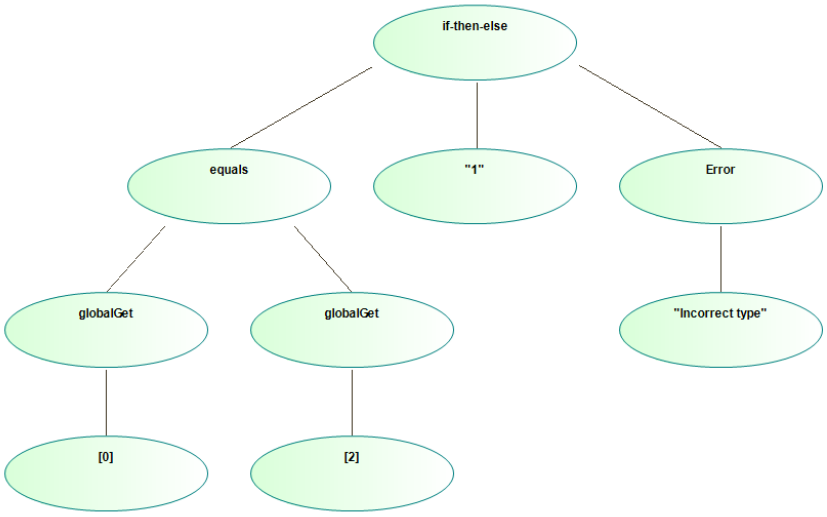

Figura 2-7 Árbol que comprueba la validez de tipos en una operación

Las operaciones que admiten los árboles son las siguientes:

- Aritméticas: suma (add), resta (sub), multiplicación (mul), división (div) y resto (modulo).
- Comparativas: menor que (lesser), mayor que (greater) e igual (eq).
- Lógicas: conjunción (and), disyunción (or) y negación (not).
- Sobre tablas: Obtener valor de la tabla global (globalGet), obtener valor de la tabla local (localGet), guardar valor en la tabla global (globalSet), guardar valor en la tabla local (localSet).
- Sobre cadenas de caracteres: concatenación (concat).
- Conversión de tipo: de str a bool (toBool) de str a int (toInt).
- Especiales: condicional (ifThenElse) con tres argumentos: la condición, la operación then y la operación else; lanzar error (Error) con un argumento: el mensaje de error.

2.4 La clase AcideAnalyzer

La clase analyzer es la encargada de leer el archivo que contiene la descripción del analizador y distribuir el texto del archivo al constructor de cada analizador.

Además, se encarga de realizar la comunicación entre los analizadores léxico y sintáctico y formar la respuesta cada vez que se pide analizar un carácter.

Esta clase también contiene tablas que sustituyen el estado actual de los analizadores dependiendo de en qué pestaña del editor nos encontremos.

2.4.1 Lectura del archivo

El archivo que contiene la descripción del lenguaje se divide en diferentes secciones marcadas por un inicio de sección.

- "==lexical_section==": Aquí se encuentran las definiciones de los *tokens*, si dos *tokens* son reconocidos a la vez, el autómata selecciona el que se encuentra más abajo en esta lista.
- "==separators==": Igual que la sección anterior, solo que estos *tokens* no se introducen en el analizador sintáctico.
- "==syntax_section==": Donde se definen las producciones y operaciones para los analizadores sintáctico y semántico.
- "==color_section==": En esta sección se relacionan los *tokens* con sus colores. Por ejemplo: "var := red" donde var es el *token* y red el color.
- "==initial_production_section==": Solo contiene una línea que es el nombre de la producción inicial.

- "auxiliar_lex_section":Contiene definiciones de *tokens* que se usan en otras definiciones. Los *tokens* definidos que nunca se introducen en el analizador léxico

Dependiendo de la sección, se manipula el texto y se crean las estructuras de datos necesarias para la construcción de los distintos analizadores.

2.4.2 Formato de la respuesta

Por cada carácter que se lee se forma una respuesta. Esta respuesta depende de si la lectura es correcta o si se ha producido un fallo:

- Si la lectura es correcta se devuelve un array con los siguientes elementos [color del *token*, el valor del *token*, la longitud del *token*].
- Si es incorrecta se devuelve un array con los siguientes elementos ["Error", el valor del *token*, la longitud del *token*, mensaje de error].

2.5 Incorporación a ACIDE

El analizador se incorpora a ACIDE mediante listeners que reaccionan a las siguientes acciones: cambiar de pestaña y escribir texto.

También se añade al menú de configuración una ventana que permite cargar el archivo de texto que la clase AcideAnalyzer usará para crear el analizador.

Capítulo 3 – Resultados

Se ha conseguido un analizador léxico y un analizador sintáctico funcionales. El analizador semántico funciona correctamente, aunque no ha podido implementar las operaciones de guardado y restauración de tablas locales de forma satisfactoria.

Esto es debido a que el guardado se debe realizar al principio de una producción. Quizá se pudiese solucionar usando un analizador LL(1) que indique cada vez que se entra en una producción nueva y produzca que se ejecute la operación de guardado de la tabla en un stack (las operaciones de guardado y restauración están implementadas, simplemente no están en uso).

En cuanto a la incorporación del analizador a ACIDE, no se ha conseguido implementar el reemplazamiento de código, la eliminación ni la escritura en medio del programa.

Esto es debido a la falta de tiempo. Pues la mayoría se ha utilizado en producir los analizadores.

3.1 Uso del Analizador

1. Abrir menú Configuración

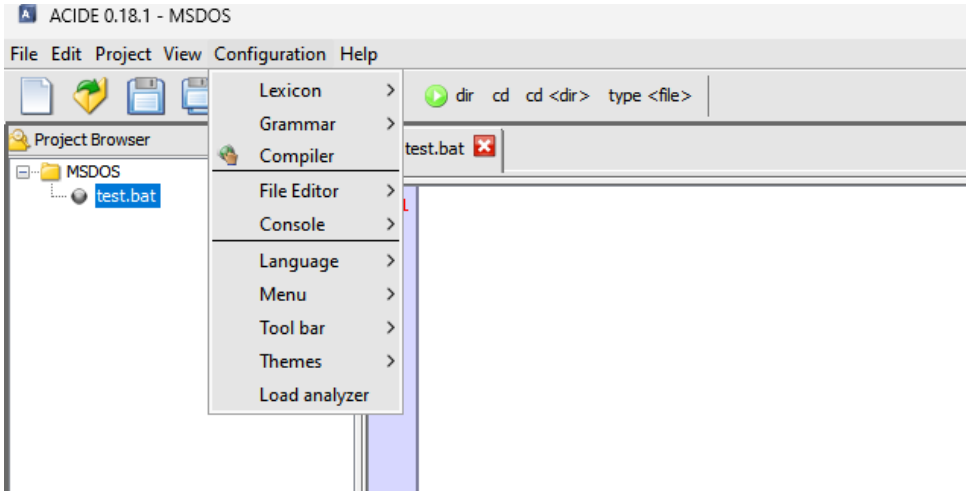

Figura 3-1 Menú de configuración

2. Seleccionar Load analyzer

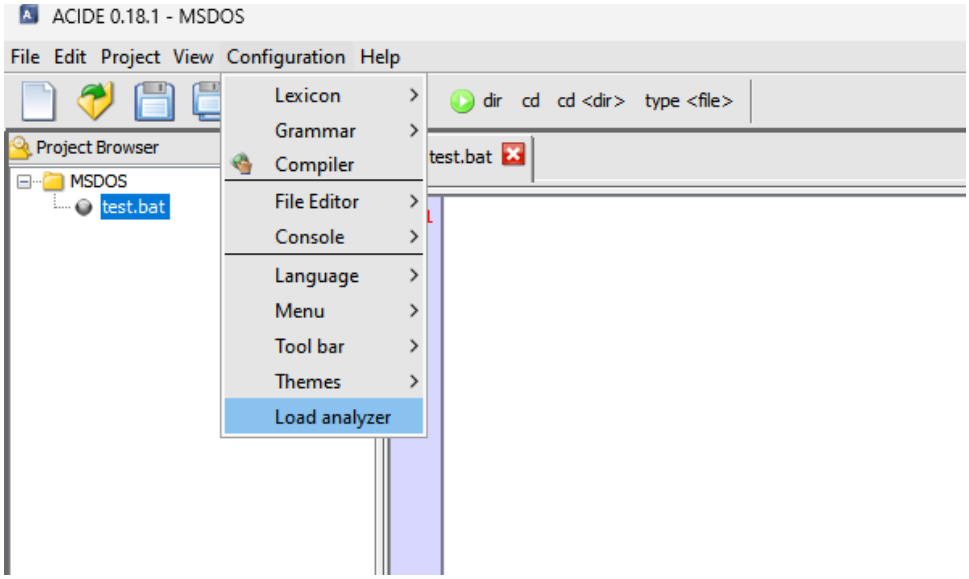

Figura 3-2 Selección de Load Analyzer

3. Cargar el archivo con la descripción del lenguaje (Se encuentra incluido con el código)

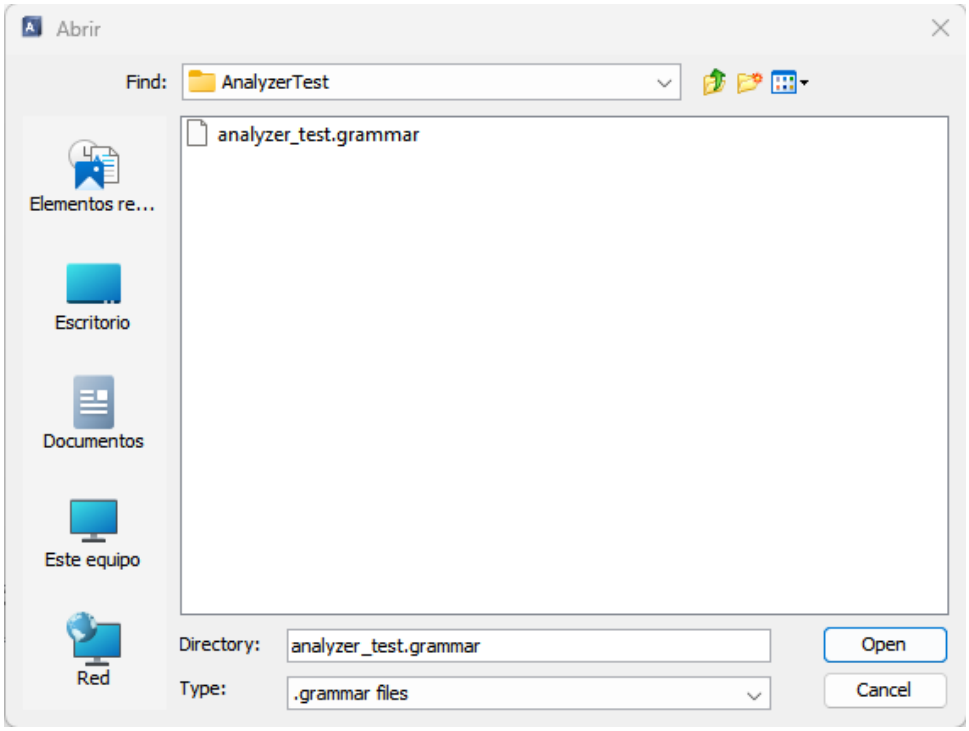

Figura 3-3 Selección del archivo del lenguaje

El archivo analyzer_test.grammar define un lenguaje dividido en dos secciones.

La primera son las definiciones compuestas por una palabra reservada (int o str), una variable, símbolo igual y un valor numérico.

La segunda consiste en una suma de una cantidad de elementos indefinida. Estos elementos han de haber sido definidos en la sección anterior.

```
==auxiliar_lex_section==
digito := 0|1|2|3|4|5|6|7|8|9
lowerletter := a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
==Lexical_section==
number := digito+
var := lowerletter+
asig := =
integer := int
string := str
add := \+
==Separators==
space := \s
eol := \n
==Syntax_section==
start := declarations operations; return := 1
declarations := declaration declarations
declarations := <epsilon>
declaration := declarator var asig number; operations := globalset([1],[0])
declarator := integer; return := int
declarator := string; return := str
operations := operations add operation; operations := eq([0],[2]); return := [2]
operations := var; return := globalget([0])
operation := var; return := globalget([0])
==Initial_production_section==
start
==Color_section==
number := green
var := blue
asig := pink
integer := yellow
string := yellow
```

Figura 3-4 Contenido de analyzer_test.grammar

4. Programar

Al reconocerse “in” como una variable se muestra de color azul.

Figura 3-5 Inserción parcial de “int”

Al introducir una “t” se reconoce “int” cuyo color se ha especificado como amarillo.

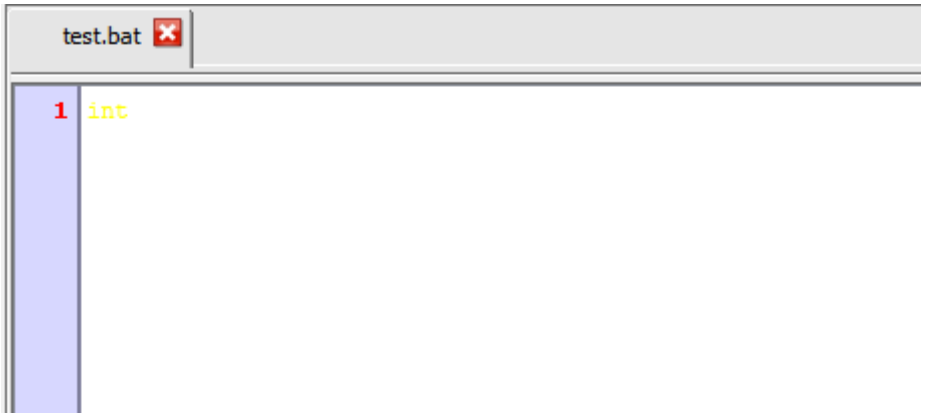

Figura 3-6 Inserción completa de “int”

Introducimos “a” que es reconocido como una variable y seguimos como un igual que es reconocido como otro símbolo y marcado por su color, rosa.

Figura 3-7 Inserción de “int a =”

Introducimos un número, al ser reconocido como tal, se marca de verde.

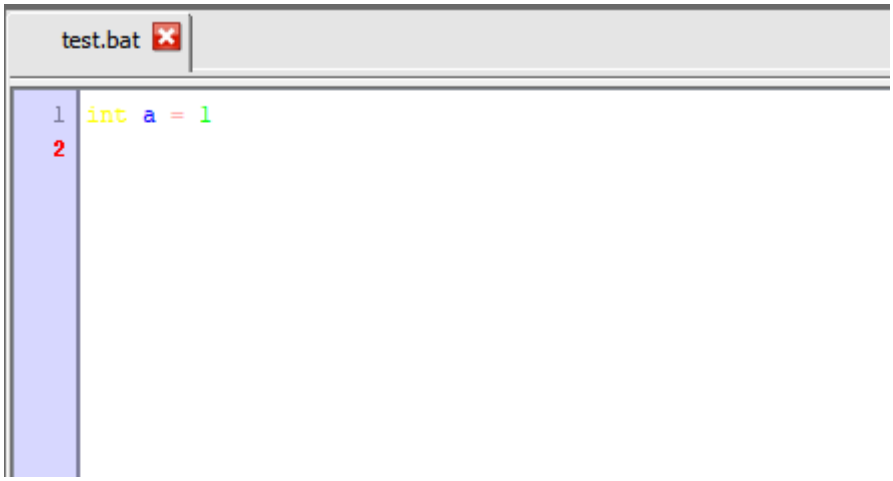

Figura 3-8 Declaración y asignación de la variable a

Definimos otra variable b igual que a por lo tanto coinciden los colores.

Figura 3-9 Declaración y asignación de las variables a y b

Ahora introducimos una suma, como a y b son variables, aparecen en azul. Como el símbolo "+" no tiene ningun color especificado, usará el color por defecto, negro.

Figura 3-10 Suma de a y b

Al introducir una nueva definición, el analizador nos devuelve un error, texto en rojo, pues el lenguaje no admite definiciones después de las operaciones.

Figura 3-11 Error, declaración después de las operaciones

Capítulo 4 – Conclusiones

Mientras que estoy orgulloso de la funcionalidad producida, y haber alcanzado el objetivo principal: crear un analizador que permita analizar un código o texto a medida que se escribe.

No puedo evitar mencionar la magnitud de un proyecto como este. A lo largo de la carrera se menciona varias veces que es más costoso modificar un programa producido por un tercero que crear uno propio. No podía hacerme a la idea de cuanto podría ser ese coste, algo que ahora tengo más claro.

ACIDE es un entorno muy potente y configurable y, aunque con dificultades, me alegro de haber participado en un proyecto así. Terminada mi aportación al proyecto espero haber estado al nivel que merece.

Enlace al código fuente

Link

Introduction

In this memory a description of the design and implementation of a compiler of compilers is made, it will be incorporated to ACIDE and used to highlight the tokens of the produced code in the ide previously mentioned and to check the syntax correction.

In the next paragraphs it will be explained in a detailed manner the implementation of the different parts of the compilator.

Motivation

During the course of my degree in Computer Engineering I adquired an interest in algorithmics and data structure, which has led me to choose ACIDE as my project for my final degree project because I consider that the task at hand falls in these fields.

The task is to give ACIDE tha ability to analyze languages described by the user and to identify errors produced when programming in that language.

ACIDE is born of the necesity of enviroments that allow the debugging of SQL databases. This ide offers a graphic interface that with DES, which allows us to carry out textual debugging, supplies this necesity. With the aim of providing more functionalities, the development of ACIDE has been continued over several years in several final degree projects.

Objectives

The aim of this project consists in the implementation and incorporation of the various functionalities belonging to a compilor capable of highlight in several colors and recognize:

- Several tokens of a language specified by the user, namely, a lexical analyzer.
- *The language syntax and its correction, namely, a syntax analyzer.*
- Optionally, identify the types and verify their correct use, namely, a semantic analyzer.

Aditionally, the analysis must be done in a progressive way, in real time, while the program is being written. For that it must be allowed, in addition to writing, code replacement, code deletion and code writing in the middle of the program.

Goals not achieved

It has been achieved the realization of the lexical and syntax analyzers and, partially, the semantic analyzer, in addition of code highlighting in several colors specified by the user.

Real time analysis has been achieved. Code replacement, code deletion and code writing in the middle of the program has not been achieved.

Project background

ACIDE is an ide (*integrated development enviroment*) that can be used as a parser, compilator or database system, like sql among others. It is constituted as a project in constant evolution due to the development of this tool by diferent groups over the years, always under the direction of Fernando Sáenz Pérez.

- During the academic year 2006-2007, by Diego Cardiel Freire, Juan José Ortiz Sánchez and Delfín Rupérez Cañas.
- During the academic year 2007-2008, by Miguel Martín Lázaro.
- During the academic year 2010-2011, by Javier Salcedo Gómez.
- During the academic year 2012-2013, by Pablo Gutiérrez García-Pardo, Elena Tejeiro Pérez de Ágreda and Andrés Vicente del Cura.
- During the academic year 2013-2014, by Juan Jesús Marqués Ortiz, Fernando Ordás Lorente and Semíramis Gutiérrez Quintana.
- During the academic year 2014-2015, by Sergio Domínguez Fuentes.
- During the academic year 2019-2020, by Sergio García Rodríguez.
- During the academic year 2020-2021, by Carlos González Torres.

Because of the kind of development of ACIDE, some standards has been established that must be followed in ACIDE code developmet.

Source code standards

As mentioned before, ACIDE is a project developed by various groups over some time. To facilitate code maintainability and future implementation of new functionalities, it has been necesary to establish some standards that should be followed by the developers;

- All the comments and identifiers must be declared in English.
- At the beginning of each class the license must be indicated: GPLv3.

```
/*
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */
```

Figura 1-1 License GPLv3

- Use of commentaries in each function and attribute indicating their purpose.
- Each java class will have a javadoc comment with this format:

```
/**
 *      Descripción de la clase
 *
 *      @version version
 * @see <NombreDeClase/NombreDeInterfaz>
 */
```

Figura 1-2 Class comment in Javadoc

- All the class attributes start with "_". For example: "_transitionTable"
- Class denomination starts with "Acide" followed by its name. For example: "AcideLexAnalyzer".
- All the name words must start in capital letters so much in attributes: "_transitionTable", classes "AcideLexAnalyzer" or methods: "applyEpsilonClosure()".
- Constant variables will have their complete name in capital letters and if it is made up by several words, these must be separed by "_". For example: "EMPTY_TRANSITION".
- All classes that create application must include these methods:

```
// Builds the ACIDE - A Configurable IDE configuration window components
private void buildComponents() {}

//Adds the components to the ACIDE - A Configurable IDE to the
//configuration window
private void addComponents() {}

//Sets the text of the ACIDE - A Configurable IDE class components
//with the labels in the selected language to display
private void setTextOfMenuComponents() {}

//Updates the ACIDE - A Configurable IDE class components
//visiblilty with the menu configuration
private void updateComponentsVisibility() {}

//Sets the listeners of the configuration window components
private void setListeners() {}
```

Figura 1-3 Mandatory methods for new windows

Work plan

At the start of the academic year Fernando Sáenz Pérez, project director, gave me access to previous years memories, ACIDE's manual and its source code.

With this material (and in the absence of studying PL) the project was started with a requerimets identification, that would end in the objectives previously described.

Later on, a brainstorm was done on how to address the diferent needs to satisfy in the project.

Later, (while studying PL) a more rigorous methodology about the project resolution was established, it consists of identifying a problem, if it is trivial, solve it, or otherwise divide it into simpler problems. Repeat until project design is done.

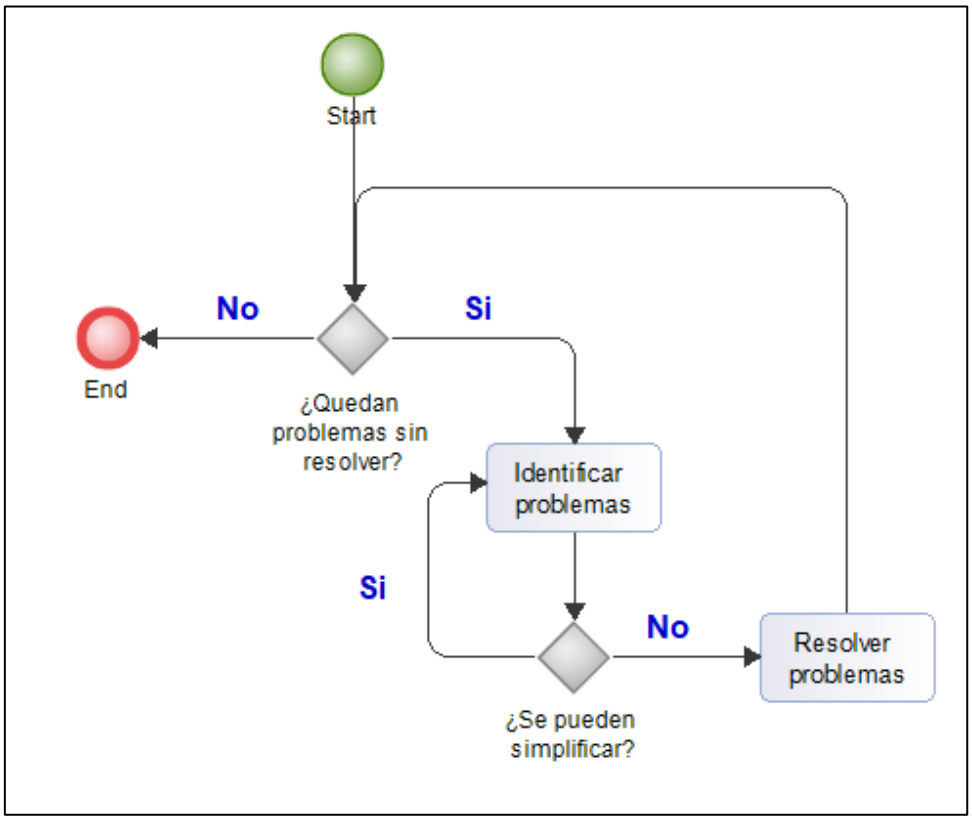

Figura 1-4 Work plan flow diagram

Once the design is done, it was time to implement it, debug it and if bugs were found, fix them using the already explained method.

Once the analyzer was finished, it was incorporated into the ACIDE system. Said incorporation was done by the identification of code to modify and adding windows where necessary so that the original code had a minimal number of modifications when at the same time a new functionality was added.

Conclusions

While I'm proud of the produced functionality, and reaching the main objetive: to make an analyzer that allows us to analyze code as it is being written.

I can't help but mention the magnitude of a project like this one. Throughout the university degree I've been told several times that is more expensive to modify an already made program done by a third party than to make one. I couldn't get an idea of how much that could be, something that now I know better.

ACIDE is a very powerful and configurable enviroment and, even with difficulties, I'm glad I participated in a project like this one. Finished my contribution to the project I hope I have lived up to what it deserves.

Bibliografía

1. Hopcroft J.E. Introducción a la Teoría de Autómatas, Lenguajes y Computación. Pearson Educación; 2005.
2. Wheeler, Scott M.L. Programming language pragmatics, 3a ed. Morgan Kaufmann;2009.
3. Manual de usuario de ACIDE, versión 5.0
4. Java api docs
5. w3schools
6. Stack overflow