Sistemas Informáticos
Curso 2005 / 2006

Transformación asistida de programas funcionales

Autores:
Elena Akinfieva
Álvaro Navarro Iborra
Erika Elizabeth Romero Sanguña

Dirigido por:
Prof. Cristóbal Pareja Flores
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

ÍNDICE

Autorización

Los abajo firmantes, Elena Akinfieva, Álvaro Navarro Iborra, Erika Elizabeth Romero Sanguña, autores del proyecto **Transformación Asistida de Programas Funcionales** en la asignatura de Sistemas Informáticos, autorizan a la Universidad Complutense de Madrid a difundir y utilizar los contenidos de este proyecto, así como el código, prototipos o documentación asociada a dicho proyecto, con fines exclusivamente académicos, nunca comerciales y mencionando expresamente a sus autores.

Participantes:

Elena Akinfieva Álvaro Navarro Iborra Erika Romero Sanguña

Fecha: Julio, 2006

Resumen

El proyecto está basado en el desarrollo de software con el modelo transformacional. Según este modelo se manipula y transforma un programa fuente inicial en un determinado lenguaje de programación, para obtener un programa final transformado. El lenguaje de programación del código fuente y del código objeto en este proyecto es el mismo.

Este proyecto está orientado particularmente a la transformación de programas funcionales. Para la construcción del sistema de transformación se requiere, además, un lenguaje de transformación para que el usuario pueda expresar los pasos de transformación que desea aplicar sobre el programa. Los lenguajes de programación funcionales que se han considerado para implementar el sistema son Haskell y ML.

Las transformaciones que se aplican en el sistema están basadas principalmente en el sistema de plegado-desplegado de Burstall y Darlington. El objetivo del modelo de software transformacional es poder obtener programas más eficientes o poder demostrar teoremas. Para ello se pueden usar algunas tácticas de transformación como: fusión, deforestación, tuplamiento, paralelización, etc.

Palabras clave: transformación de programas, programación funcional, optimización, pliegue/despliegue, Haskell, ML.

Abstract

This project is based on the development of the software with the transformational model. According to this model an initial source program in a certain programming language is manipulated and is transformed with the aim to obtain a final transformed program. The programming language of the source code and of the object code is the same in this project.

This project is particularly oriented to the transformation of functional programs. Also, for the construction of the transformational system the transformation language is needed, so that the user would be able to write the steps of transformation that he wishes to apply onto the program. The functional programming languages which have been considered for the system implementation are Haskell and ML.

The transformations that are applied in the system are based, mainly, on the fold/unfold system of Burstall and Darlington. The objective of the transformational model of software is to obtain more efficient programs or to prove theorems. For this purpose, different transformation tactics can be used, as, for example, fusion, parallelization, tupling, deforestation, etc.

Keywords: program transformation, functional programming, optimization, fold/unfold, Haskell, ML.

Agradecimientos

En primer lugar, queremos agradecer la ayuda y la comprensión de nuestras familias, que nos han apoyado, animado y soportado durante todo el año.

Agradecemos a nuestro profesor Cristóbal Pareja Flores su apoyo e interés por el proyecto y la dedicación que nos ha mostrado.

Finalmente, agradecemos a todos nuestros compañeros y amigos que nos han acompañado y animado durante esta travesía.

Gracias a todos.

1. INTRODUCCIÓN

El objetivo principal que persigue este proyecto es realizar un sistema que a partir de un programa de partida **P** genere un programa transformado **P'**.

Dicho programa transformado **P'** resuelve el mismo problema y es semánticamente equivalente a **P**, pero que goza de mejor comportamiento respecto a cierto criterio de evaluación. En la literatura se han propuesto una gran variedad de transformaciones para mejorar el código, entre las que destacan las transformaciones de plegado/desplegado ("folding/unfolding") y la evaluación parcial de programas.

En el proyecto realizaremos la transformación de programas mediante las reglas del sistema de pliegue/despliegue que describiremos en profundidad a lo largo de la memoria. Estas reglas fueron formuladas originalmente para la programación funcional en 1977 por Burstall y Darlington [BD77] y posteriormente introducidas en la programación lógica por Tamaki y Sato [TS84].

Lo que se ha hecho en este proyecto es un sistema que pueda ser capaz de reconocer programas funcionales y aplicarles varias reglas de transformación para mejorar su eficiencia.

Se ha conseguido un sistema con un entorno amigable y fácil de utilizar que realiza todas las reglas de transformación.

Hasta ahora se han realizado algunas aplicaciones informáticas en las que se ha planteado el modelo transformacional. La mayoría de ellas tienen ciertas limitaciones o en algún caso son sistemas en los que no se ha llegado a desarrollar completamente su implementación, aunque también hay algún sistema bastante potente capaz de aplicar técnicas complejas de transformación.

Algunos de estos sistemas son:

STARSHIP [YFPG]: permite hacer demostraciones de propiedades para luego poderlas aplicar. En particular el sistema de inferencia basado en el principio de inducción estructural.

HERA [HERA] (Haskell Equational Reasoning Assistant): El sistema fue proyectado por Andy Gill bajo la dirección de Peyton Jones. Aunque es un proyecto inacabado posee varias ventajas, entre las cuales tenemos: interfaz gráfica, aplicación de leyes, principio de demostración de propiedades basada en el principio de inducción estructural.

PROXAC: Este sistema, proyectado por Jan Van Snepchentz, trabaja con notación BMF (Bird-Meertens Formalism). Este sistema está especialmente pensado para "Teoría de listas" de Bird [B87].

MAP [MAP]: Es un sistema de derivación de programas basado en reglas de transformación y estrategias. Se basa en las reglas de transformación de Burstall y Darlington. A diferencia de otros sistemas, MAP transforma programas lógicos escritos en PROLOG.

MAG [MAG]: Es un sistema que permiten hacer encaje de segundo orden restringido usando técnicas de búsqueda como el backtracking.

La mayoría de estos sistemas permiten hacer transformaciones con el sistema de pliegue-despliegue y encaje de patrones simples, pero hay algunos sistemas como el MAG que son más complejos, con encaje de segundo orden restringido.

A continuación, describimos a modo de resumen cómo está estructurada la memoria y qué se incluye en cada uno de
los apartados:

Apartado 2: Planteamiento inicial, requisitos, decisiones de diseño, logros y limitaciones del proyecto.

Apartado 3: Análisis de requisitos, Diseño, Modelo de desarrollo, Ciclo de vida y Plan de pruebas del sistema.

Apartado 4: Descripción, características limitaciones y gramática del lenguaje fuente.

Apartado 5: Definición, requisitos a nivel de implementación, formato y descripción detallada de las reglas de transformación del lenguaje de transformación.

Apartado 6: Define cada una de las tácticas de transformación, como por ejemplo, deforestación y tuplamiento entre otras.

Apartado 7: Describe la arquitectura y estructura de la aplicación.

Apartado 8: Describe las herramientas utilizadas para la implementación de la aplicación.

Apartado 9: Descripción de varios sistemas existentes que realizan transformaciones de lenguajes funcionales y muestra la apariencia de los más significativos.

Apartado 10: Conclusiones finales del proyecto y posibles ampliaciones para un trabajo futuro.

Apartado 11: Varios ejemplos de las diferentes reglas de transformación y ejemplos de la aplicación de técnicas sobre programas concretos.

Apartado 12: Bibliografía detallada del las diferentes fuentes en las que se ha consultado para la realización de este sistema.

Además de la memoria, se he realizado una página web, que tiene una breve descripción de la herramienta y se encuentra en la siguiente dirección:

http://aljibe.sip.ucm.es/akrona

Tiene los siguientes apartados:

- Home
- Descripción
- Manual de usuario
- Ejemplo de uso
- Tácticas
- Links relacionados
- Contacto

2. PLANTEAMIENTO DEL PROYECTO

2.1. Planteamiento inicial

Se propuso desarrollar un sistema con que manipular formalmente programas funcionales. Se parte de un programa ya hecho y el sistema permite al usuario indicar dónde y cómo ha de aplicar una cierta regla, transformando el programa en otro equivalente. Para ello, se han de considerar dos lenguajes, el lenguaje fuente y el lenguaje de transformación.

En el planteamiento inicial sobre el sistema de reglas aplicables a nuestro programa mediante nuestro lenguaje de transformación era el descrito en el sistema Burstall y Darlington [BD77]. Para usar dichas reglas en nuestra aplicación debíamos definir la sintaxis de cada una de las transformaciones.

Sobre el lenguaje fuente el único requisito a priori era que se tratase de un lenguaje funcional de amplia definición. Lo que no estaba definido como una exigencia propia del proyecto era el lenguaje de programación que el usuario debía usar para definir los programas funcionales de partida.

2.2. Cambios con respecto al planteamiento inicial

En el primer planteamiento no estaban detallados los aspectos referidos a la implementación del sistema.

Para diseñar el lenguaje fuente se analizaron principalmente los lenguajes Haskell y ML (Meta-Lenguaje). Después del estudio de ambos lenguajes se optó por un subconjunto del lenguaje de ML aunque sin renunciar a usar algunas características de Haskell, que no estaban incluidas en ML. Las características de Haskell incluidas en el lenguaje son el uso de patrones en la parte izquierda de las funciones, concretamente el patrón sucesor para los naturales y los dos patrones para las listas, el de lista vacía y el de lista no vacía. Además del uso de definiciones locales con *where*, ya que la gramática orignal de ML usa *let*.

Como había aspectos muy amplios a considerar en el lenguaje se decidió no tomar todo el lenguaje ML sino un subconjunto de ML acotando varios aspectos de la gramática del lenguaje. En el subconjunto de ML no se han incluido los módulos ni los tipos.

Los programas hechos a los que se les quiere aplicar las transformaciones, tienen que adaptarse ahora a ese lenguaje mezcla de Haskell y ML.

2.3. Plataforma y herramientas empleadas.

Las herramientas necesarias para este sistema son:

- Un Pc con Windows, Linux o Solaris.
- Programas de libre distribución como Java y JavaCC. El entorno de desarrollo de Java debe tener un compilador jdk_1.4.2. Para la implementación de este sistema se ha utilizado JBuider 9.0 y JavaCC4.0, pero se puede usar cualquier entorno JAVA.

2.4. Decisiones de diseño

Las primeras decisiones de diseño han sido la elección de la plataforma donde se va a usar la aplicación, el lenguaje de programación en el cual vamos a desarrollar la aplicación y las herramientas que vamos a usar para ayudarnos a construir la aplicación.

En primer lugar se ha tenido que decidir qué sistema operativo se va a usar para la realización del proyecto. En LINUX se tenía la ventaja de que existía un compilador de Haskell hecho en LINUX, y así se podría reutilizar una gran parte de código. Pero a la vez teníamos el inconveniente que ninguno de los miembros del proyecto tenía conocimiento de esta herramienta, lo que hubiera supuesto emplear tiempo en el aprendizaje de la misma. Además el implantar toda la aplicación en Haskell tiene el inconveniente de que no existen apenas programas que permitan crear entornos gráficos con Haskell.

WINDOWS tenía la ventaja de que es el sistema operativo más extendido en todo el mundo y todos los componentes del grupo ya lo usábamos habitualmente, al contrario que LINUX, que no lo usábamos con mucha frecuencia. Por tanto, el optar por WINDOWS, suponía que por un lado no se ha invertido tiempo en el aprendizaje de herramientas de LINUX, y por otro el poder contar con nuestros conocimientos en programación en entornos que corren sobre Windows.

La segunda decisión a tomar era la elección del lenguaje en el que se iba a implementar la aplicación. Después de considerar varias posibilidades, se han seleccionado dos: Haskell y Java. Haskell es un lenguaje más conciso, pero más difícil de depurar, además de la ya mencionada dificultad de crear interfaces gráficos, y la experiencia de desarrollo de aplicaciones por parte de los componentes del grupo en Haskell es muchísimo menor que en Java. Por otro lado, habríamos tenido facilidades en el sistema de transformación.

JAVA es el lenguaje que más hemos utilizado y de fácil depuración. El interfaz es fácil de programar y se obtienen entornos gráficos más vistosos que los obtenidos con herramientas de Haskell para crear interfaces. Por todo lo anterior se decidió usar Java.

En cuanto a las herramientas para procesar el lenguaje fuente se estudiaron varias alterativas como YaCC, Lex y JavaCC. Finalmente se optó por JavaCC debido a que era la más potente de todas y además el hecho de que se decidió que nuestra aplicación fuera en Java facilitaba la integración del parser para ML construido con JavaCC con el resto de módulos de la aplicación.

2.5. Logros y limitaciones

Entre los logros tenemos:

- Definición programas fuente que pueden tener expresiones, aplicación de funciones, instrucciones de selección (if, case), definiciones locales (where) y composición de funciones.

- Análisis de programas funcionales mediante nuestro analizador sintáctico, incluyendo la detección de errores en caso de que los programas sean incorrectos. Si se produce error se indica al usuario mediante un mensaje donde se ha producido el error.

- Definición de patrones, como listas, tuplas, variables y constantes.

- Ejecución de todas las reglas de transformación.

- Inclusión de un amplio repertorio de ejemplos de utilización de cada una de esas reglas y, por otro lado, ejemplos donde se muestran varias técnicas de transformación.

- Evaluación perezosa, es decir, las expresiones no se evalúan hasta que no se necesitan los resultados.

- Reducción de expresiones simples.

- Interfaz agradable y cómodo de utilizar, que contiene ventanas en las que se podrá ver paso a paso cada una de las transformaciones que se le aplicará a un programa inicial. Consta también de un sistema de ayuda y posibilidad de poder ver el árbol sintáctico del programa en tratamiento.

- Posibilidad de cargar ***scripts*** con una secuencia de instrucciones de transformación de manera automática o de manera manual siendo el usuario el que indique cuándo quiere ejecutar un paso.

Entre las limitaciones del sistema de transformación están las siguientes:

- En las tuplas, el encaje de patrones sólo puede ser simple. Esto no es una limitación en realidad, ya que se puede hacer encaje en dos pasos. De hecho, desde el punto de vista metodológico, es mejor proceder a efectuar las transformaciones paso a paso.
- No tenemos implementada una regla de inferencia para la inducción estructural y no tenemos reglas condicionales.
- No tenemos manipulación de propiedades, lo cual es debido a que el miembro izquierdo no puede ser una expresión arbitraria.

Algunas de estás limitaciones son bastante complejas, como puede ser la implementación de un sistema de inferencia de tipos. El resto podrían ampliarse de manera menos costosa y mejoraría la potencia de la aplicación, como comentaremos en las conclusiones.

3. METODOLOGÍA DEL DESARROLLO

3.1. Análisis de Requisitos

Requisitos funcionales

Los requisitos funcionales son:

Análisis sintáctico de programas funcionales: analiza sintácticamente un programa funcional y genera su árbol sintáctico correspondiente.

Aplicación de varias reglas de transformación: el sistema permite aplicar las reglas de transformación mediante su instrucción correspondiente (Véase el apartado 5.4.).

Definición de leyes: durante la transformación el usuario puede definir leyes propias y aplicarlas posteriormente.

Utilización de leyes y librerías de funciones: se pueden aplicar leyes, ya sean predefinidas o definidas por el usuario, en la transformación de un programa. También se pueden importar funciones de las librerías predefinidas al programa en transformación.

Detección de errores sintácticos: en caso de que el analizador sintáctico detecte un error, se le informa al usuario de ello, y no genera el árbol sintáctico.

Detección de errores del formato de las reglas de transformación: en caso de que el formato de una regla de transformación sea incorrecto, el sistema lo detecta y muestra el mensaje correspondiente.

Detección de errores de ejecución: en caso de haber una operación incorrecta, el sistema lo detecta y no permite la ejecución de la misma, por ejemplo, la división por cero.

Historial de reglas aplicadas: el sistema lleva un historial de las reglas de transformación aplicadas, y permite al usuario deshacer tantas reglas como desee. El historial se refleja en el árbol sintáctico del programa a transformar. Ese árbol se tiene que mantener actualizado, a medida que se aplica una transformación.

Requisitos del interfaz

Los requisitos del interfaz son:

Cargar programa: carga un programa escrito en nuestro lenguaje funcional, para poder aplicarle posteriormente las transformaciones que se deseen.

Nuevo: elimina el programa fuente y todas las transformaciones aplicadas y limpia el interfaz para poder realizar una nueva transformación.

Fijar programa de partida: realiza el análisis sintáctico del programa de partida cargado generando su árbol sintáctico, en dicho árbol se pueden aplicar todas las reglas de transformación.

Ver Árbol Derivación: permite visualizar gráficamente el árbol de derivación del programa actual que estamos transformando. Esta opción ha sido un herramienta muy útil para realizar la aplicación, y tendría bastante utilidad si se realizaran ampliaciones del proyecto.

Ver Árbol Derivación2: permite visualizar el árbol de derivación, de la misma manera que el anterior, pero para un programa secundario que se forma con algunas ecuaciones descartadas durante el proceso de transformación.

Ayuda: permite al usuario obtener ayuda tanto de como realizar el proceso de transformación en la aplicación como ejemplos concretos con cada una de las reglas de transformación.

Salir: permite finalizar la aplicación.

Requisitos de recursos

Espacio libre: 4MB
CPU: 350 MHz
Memoria: 64 MB.
Sistema operativo: Windows 98 o superior.

Para poder ejecutar la aplicación es necesario tener instalada la Máquina Virtual de Java. Los requisitos de memoria y velocidad del procesador especificados arriba son orientativos, ya que la aplicación consume muy pocos recursos. La aplicación de cada regla es imperceptible del orden de décimas de segundo. La única fase que se realiza en un tiempo más elevado, y es más perceptible por el usuario (2-3 segundos) por la razón de cargar las librerías es cuando hay un despliegue de una función muy compleja.

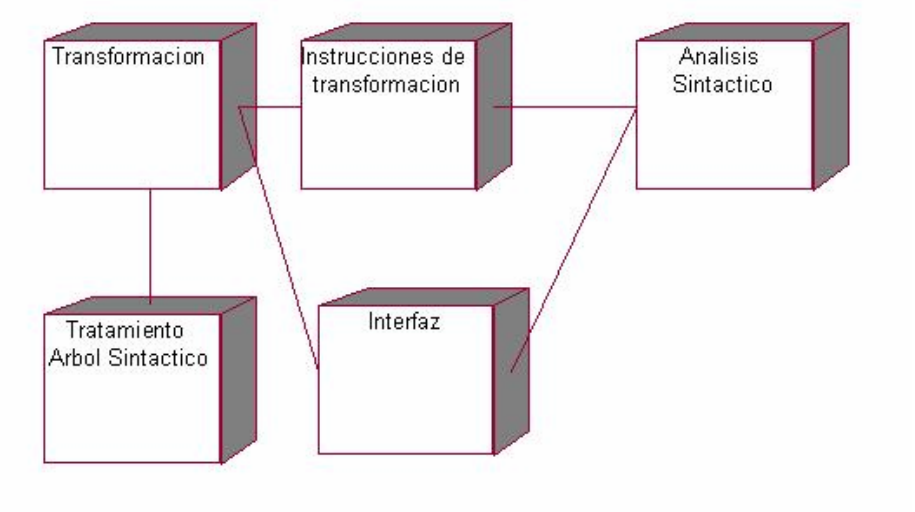
Requisitos de calidad

Se cumplen parcialmente los objetivos especificados. La aplicación es capaz de realizar transformaciones completas como vemos en el repertorio de ejemplos. No obstante se podrían añadir mejoras para que la aplicación pudiera manejar transformaciones de mayor complejidad.

3.2. Diseño

3.2.1. Arquitectura del sistema

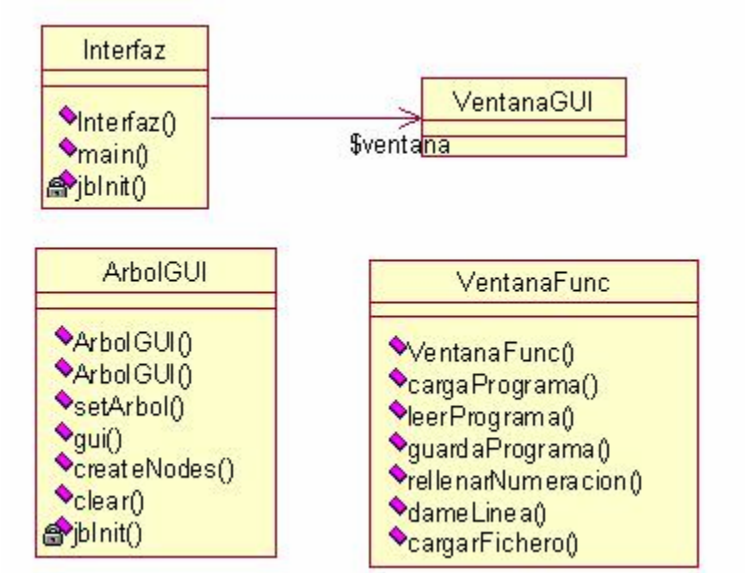
Véase el apartado 7.2. Descripción de la arquitectura.



3.2.2. Diagramas de clases

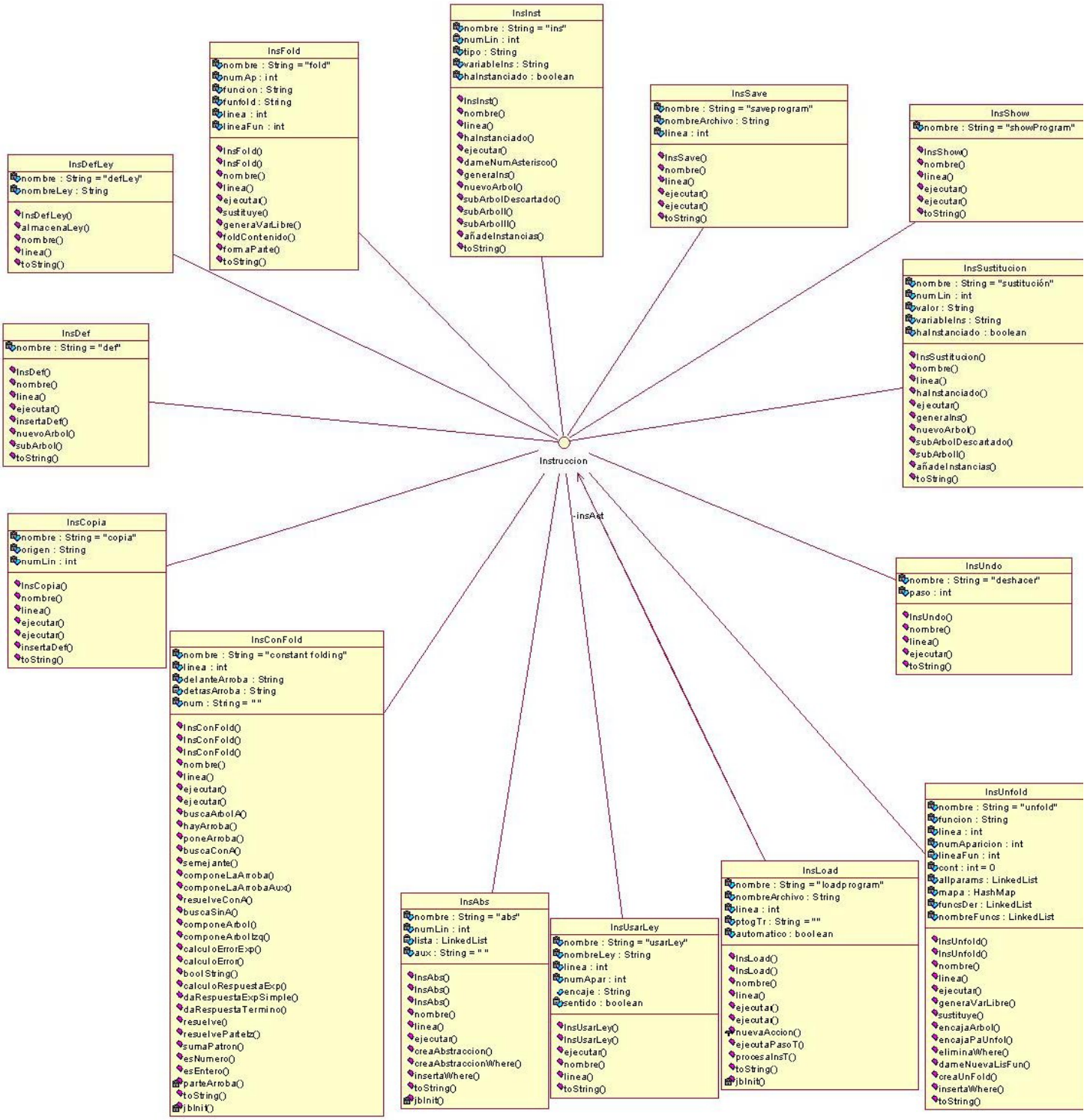
3.2.2.1. Módulo del interfaz

Véase el apartado 7.2.4. Interfaz.



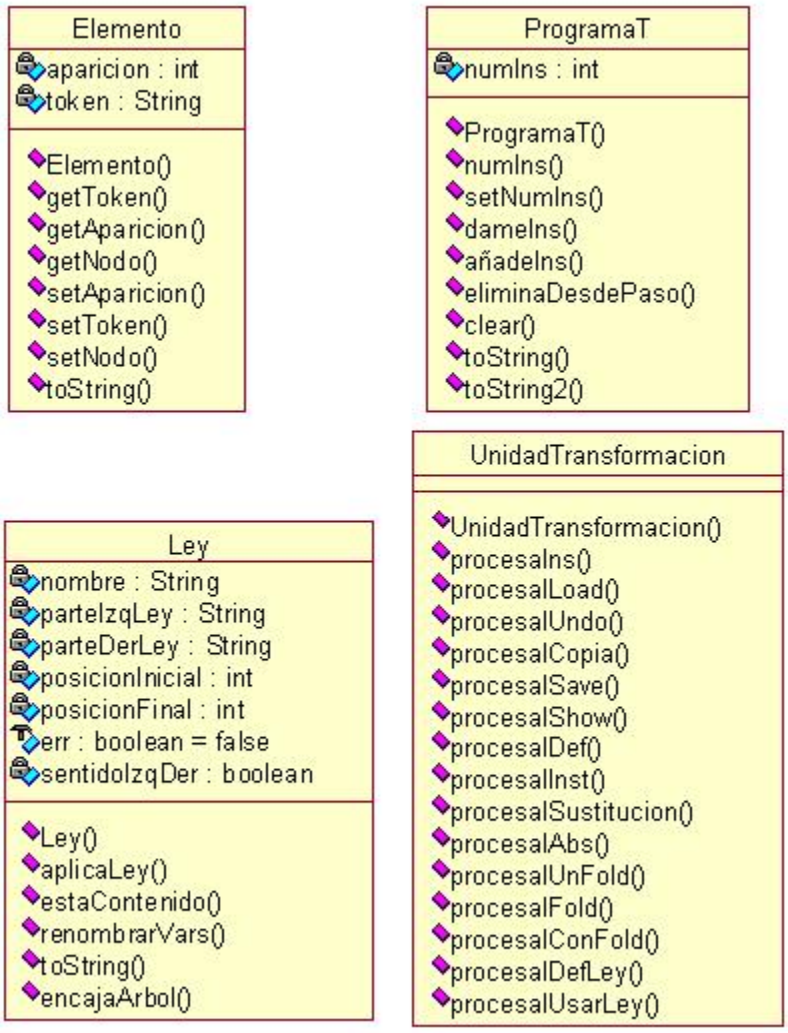
3.2.2.2. Módulo de las instrucciones

Véase el apartado 7.2.3. Módulo de transformación.



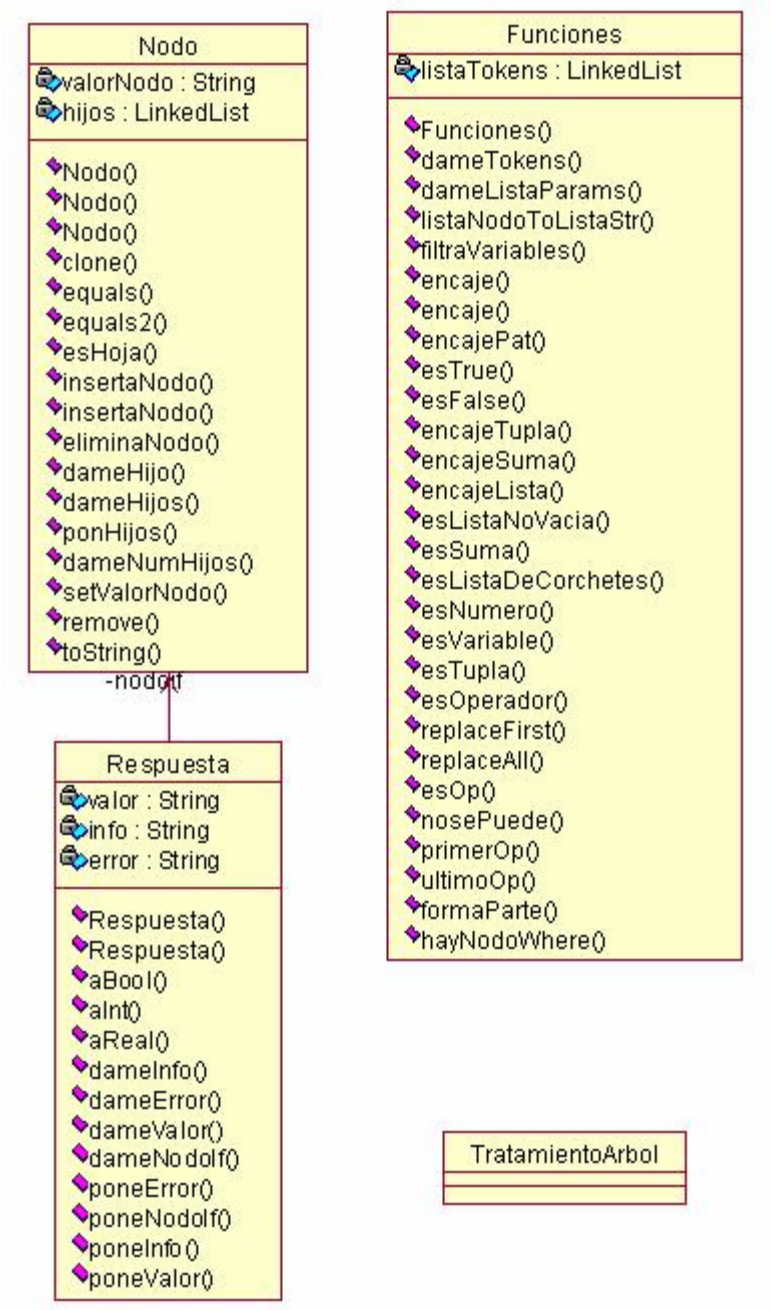
3.2.2.3. Módulo de las transformaciones

Véase el apartado 7.2.3. Módulo de transformación.

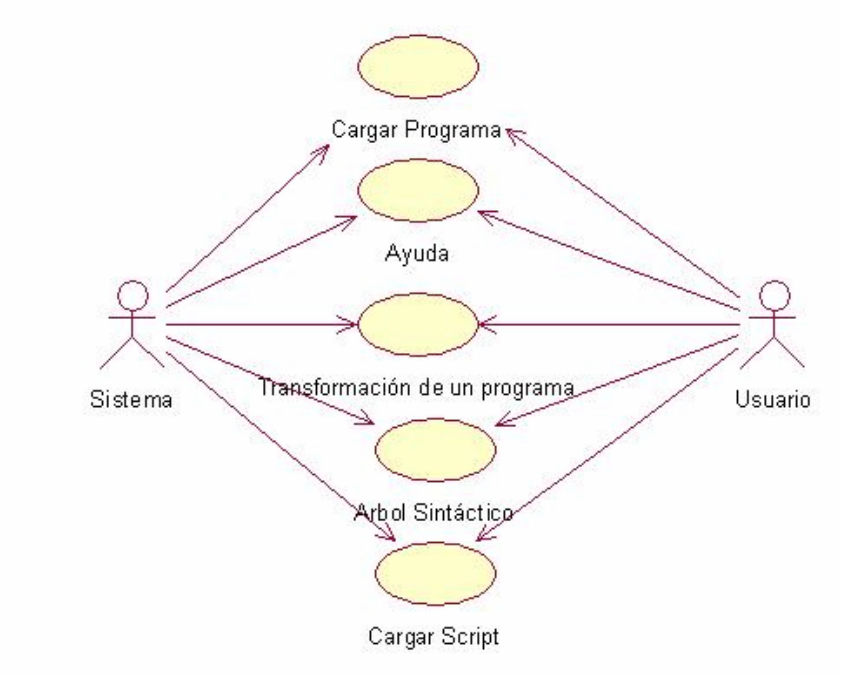


3.2.2.4. Módulo del tratamiento del árbol sintáctico

Véase el apartado 7.2.2. Módulo de tratamiento del árbol sintáctico.



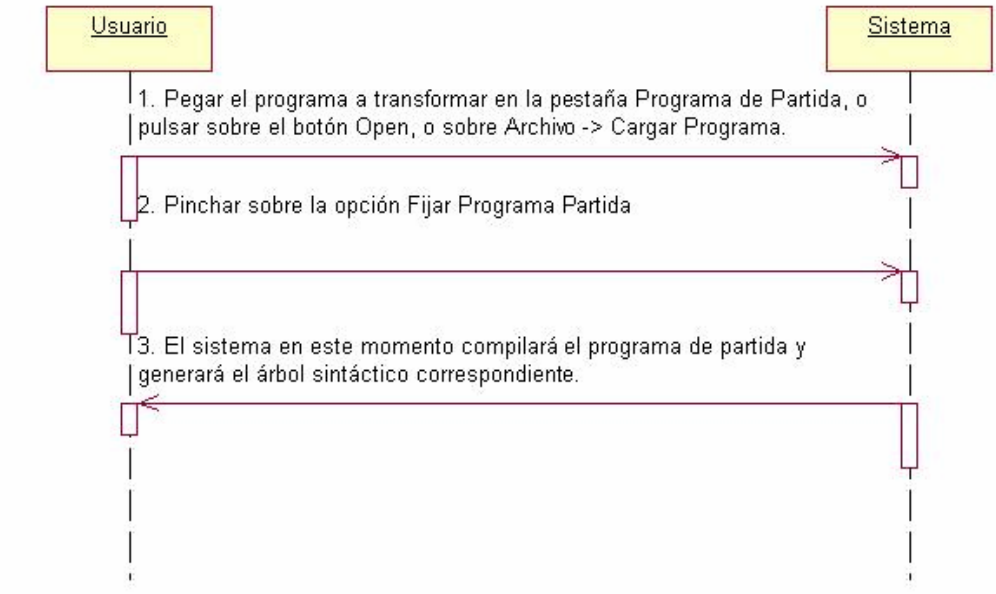
3.2.3. Diagrama de los casos de uso



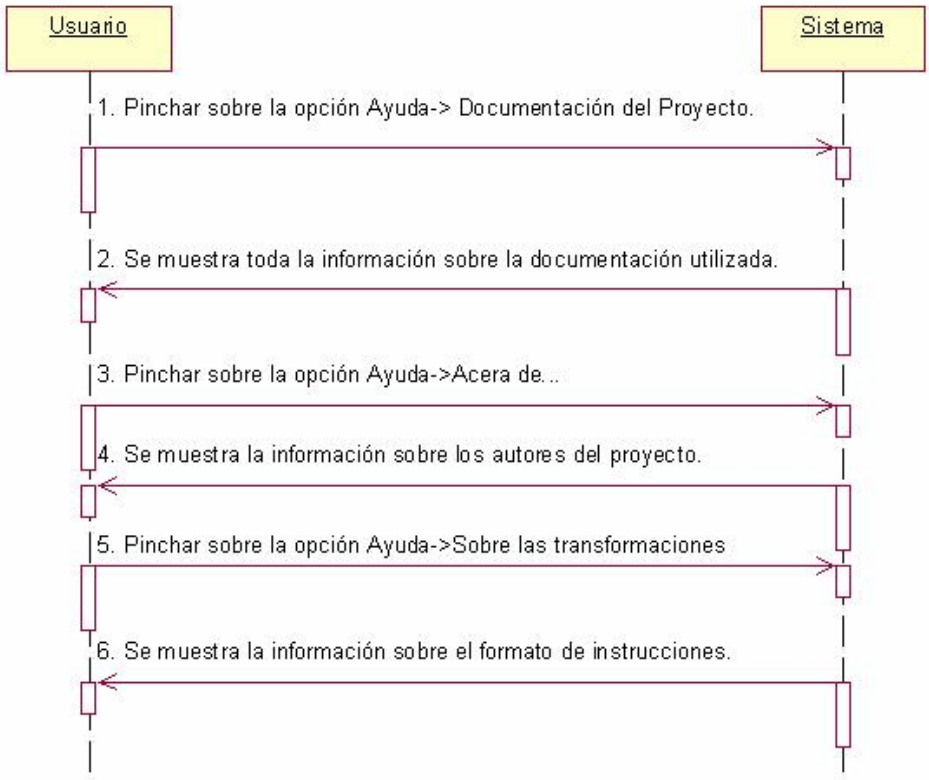
3.2.4. Diagramas de cada uno de los casos de uso

Cada uno de los casos de uso son operaciones simples detalladas.

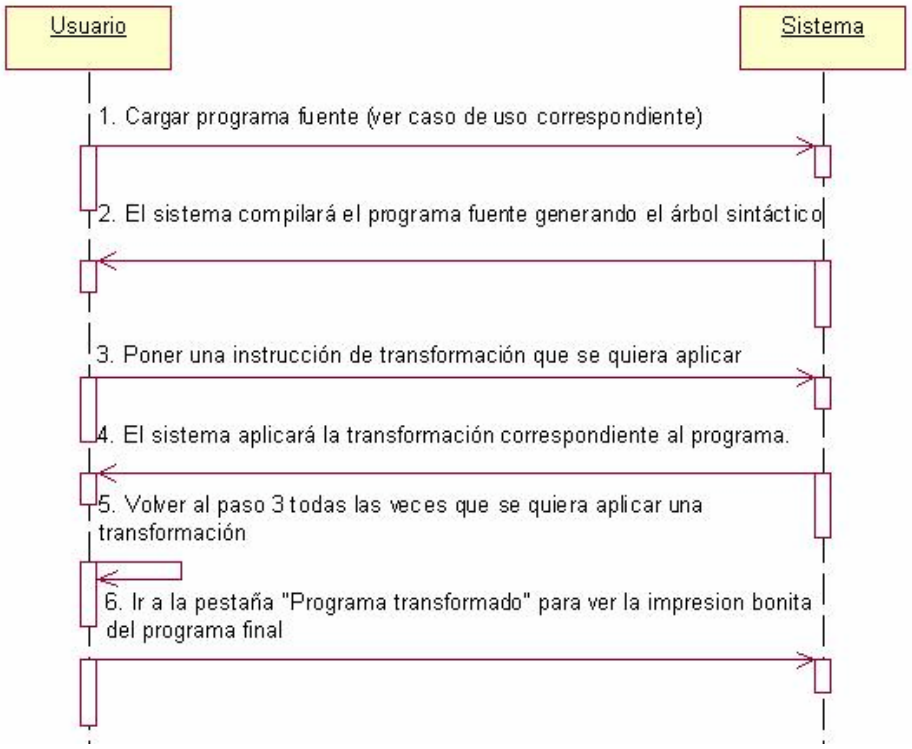
3.2.4.1. Caso de uso: Cargar Programa



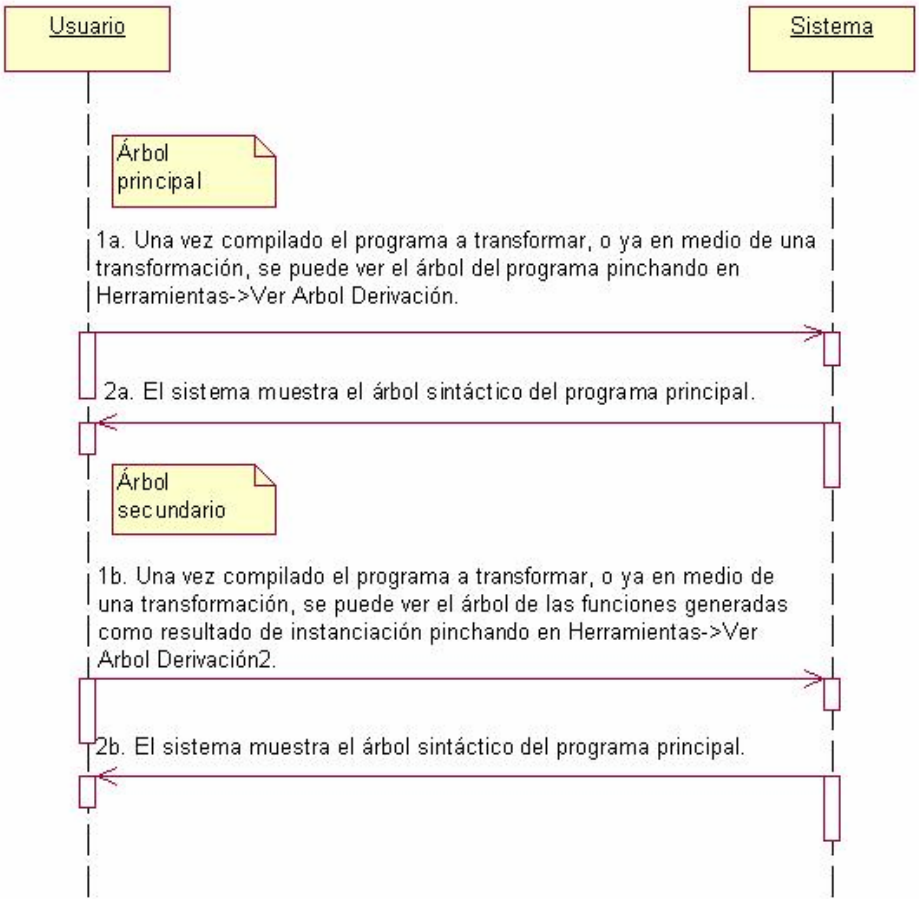
3.2.4.2. Caso de uso: Ayuda



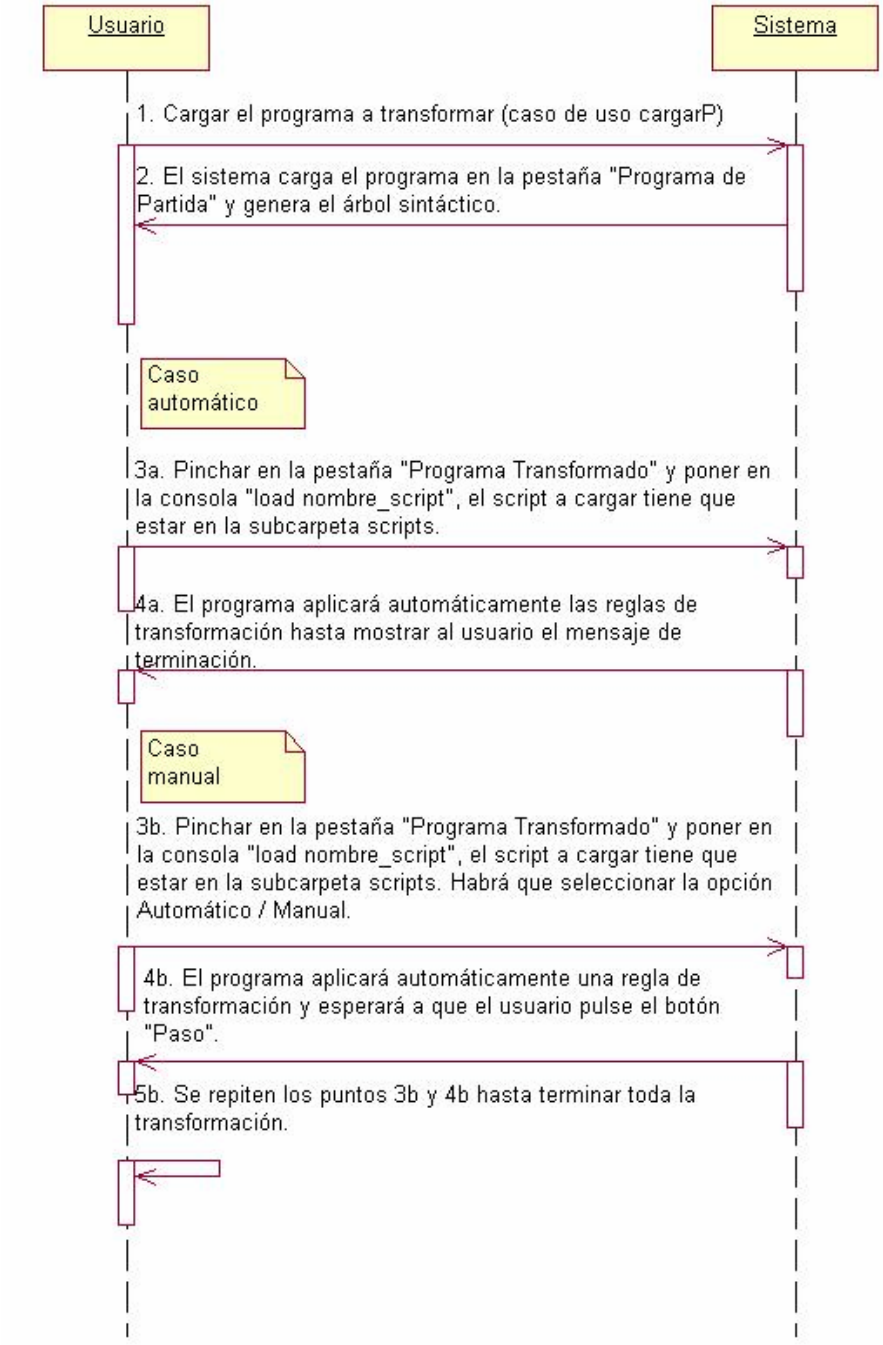
3.2.4.3. Caso de uso: Transformación de un programa



3.2.4.4. Caso de uso: Árbol sintáctico



3.2.4.5. Caso de uso: Cargar un script



3.3. Modelo de desarrollo

Se ha optado por un modelo de desarrollo mediante eXtreme Programming. Desde el primer momento se partió de la idea básica del sistema de transformación que se quería implementar y las partes básicas del sistema, a partir de esto se empezó a implementar el sistema.

La programación extrema o ***eXtreme Programming*** (XP) es una aproximación a la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, ***Extreme Programming Explained: Embrace Change***. Se trata de un proceso ágil de desarrollo de software.

Las características de este modelo que se han aplicado en el proyecto son:

- Desarrollo iterativo e incremental. Pequeñas mejoras, unas tras otras.

- Pruebas unitarias continuas, frecuentemente repetidas y automatizadas.

- Programación por parejas. Las tareas de desarrollo se han realizado casi siempre por dos personas en un mismo puesto. Esto supone que la calidad del código escrito es mayor al ser revisado y discutido mientras se escribe, aunque se produzca pérdida de productividad inmediata.

- Frecuente interacción del equipo de programación con el cliente o usuario. Reuniones periódicas cada semana o cada dos semanas, para revisar los objetivos realizados y fijar nuevos objetivos.

- Corrección de todos los errores antes de añadir nueva funcionalidad. Cada vez que se ha ido ampliando el sistema se han corregido los errores que se han detectado. Se han realizado entregas frecuentes de la versión actual de la aplicación.

- Refactorización del código. En determinados momentos del desarrollo se han reescrito algunas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento.

- Propiedad del código compartida. En vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, todos los componentes del grupo han podido corregir y extender cualquier parte del proyecto.

- Simplicidad en el código. Para programar más cómodamente, hacer la aplicación lo más legible posible y poder ampliar funcionalidades de manera más sencilla.

3.4. Ciclo de Vida

La clasificación mediante iteraciones se ha realizado a posteriori para dividir las partes implementadas. De esta manera diferenciamos el trabajo realizado en cada parte de la aplicación.

En el desarrollo del proyecto se han realizado 4 iteraciones. En cada una de las iteraciones se han desarrollado partes diferentes del software del sistema que describimos a continuación.

En cada una de las iteraciones se han realizado siempre las siguientes tareas:

- Reuniones semanales con el profesor director del proyecto, para revisar las tareas realizadas y concretar nuevos objetivos.

- Planificación, para definir el tiempo, esfuerzo, recursos y disponibilidad de cada uno de los miembros del grupo.

- En cada iteración se ha realizado un plan de pruebas para cada parte implementada.

Primera Iteración (noviembre-enero)

- Estudio de los posibles lenguajes en los cuales trabajaría el sistema para hacer las transformaciones de programas funcionales.

- Estudio de diferentes herramientas de compilación para realizar el módulo de análisis sintáctico del lenguaje fuente de la aplicación.

- Implementación del módulo de análisis sintáctico del lenguaje fuente del sistema de transformación.

Segunda Iteración (febrero-abril)

- Definición de un lenguaje de reglas de transformación para transformar el programa funcional.

- Implementación del módulo del lenguaje de transformación.

Tercera Iteración (abril-mayo)

- Revisión de errores de funcionamiento sobre los ejemplos realizados de transformaciones.

- Implementación de funcionalidad carga y visualización automática paso a paso de transformaciones completas.

- Recopilación de todo el material de documentación elaborado durante el proyecto.

Cuarta Iteración (junio)

- Búsqueda de diversa documentación para la memoria.

- Elaboración de la memoria del proyecto.

- Elaboración de página Web informativa sobre los contenidos del proyecto y el funcionamiento de la aplicación.

- Corrección de los últimos aspectos de la memoria.

- Realización de presentación.

3.5. Plan de Pruebas

Se han realizado varias pruebas a lo largo del proyecto. En cada iteración se han hecho sobre las distintas partes de la aplicación.

1º) Caso de prueba del funcionamiento del parser

En la primera iteración se han realizado varias pruebas sobre el correcto análisis de los programas de partida y la correcta generación de los árboles sintácticos.
En un primer lugar los árboles se generaban con la herramienta jjtree del javacc, pero su estructura no era correcta para su uso posterior en la aplicación y al detectar este problema tuvimos que implementar una estructura de datos a medida de los algoritmos que se necesitarían para realizar las transformaciones.

2º) Caso de prueba de las instrucciones de transformación

En la segunda y tercera iteración se ha sometido cada instrucción de transformación a varias pruebas. Por un lado para detectar errores con las posibles entradas del lenguaje fuente, y por otro lado para obtener una documentación detallada y variada del uso de las instrucciones de transformación.

3°) Caso de prueba de los ejemplos de transformación

En la tercera iteración se han realizado pruebas sobre los ejemplos de transformación completos, es decir, de varias instrucciones de transformación encadenadas. Durante la realización de estas pruebas se ha ido probando que no se produzcan errores en la modificación del árbol sintáctico al encadenar varias instrucciones de transformación seguidas; y por tanto comprobando que una secuencia de instrucciones de transformación ofrece un resultado correcto.

4. LENGUAJE FUENTE

4.1. Descripción del lenguaje fuente

El lenguaje fuente usado en el proyecto es un lenguaje funcional basado en ML y en Haskell. El lenguaje de referencia que se ha elegido es ML. Hay varias versiones de ML [SML97], como, el Moscow ML [MML], Ocaml [OCAML].

ML es un lenguaje de programación de propósito general de la familia de los lenguajes de programación funcional desarrollado por Robin Milner y otros a finales de los años 1970 en la Universidad de Edimburgo. ML es un acrónimo de **Meta Lenguaje** dado que fue concebido como el lenguaje para desarrollar tácticas de demostración en el sistema LCF (El lenguaje para el cual ML era meta lenguaje es *pplambda*, una combinación del cálculo de predicados de primer orden y el lambda-cálculo polimórfico sencillamente tipificado).

Frecuentemente se clasifica a ML como un lenguaje funcional **impuro** dado que permite programar imperativamente con efectos laterales, a diferencia de otros lenguajes de programación funcional puros, tales como Haskell.

4.2. Características del lenguaje fuente

Entre las características de ML se incluyen evaluación por valor, manejo automatizado de memoria por medio de recolección de basura, polimorfismo parametrizado, análisis de estático de tipos, inferencia de tipos, tipos de datos algebraicos, llamada por patrones y manejo de excepciones.

El lenguaje fuente utilizado en este proyecto será un lenguaje funcional basado en ML y Haskell, por lo tanto consta de características propias de estos lenguajes, como por ejemplo:

- un conjunto de operaciones primitivas cuyo significado está predeterminado en el sistema. Por ejemplo, la suma de números enteros, la resta, el producto, etc.
- un conjunto de definiciones de función, establecidas por el programador, que eventualmente emplearán las operaciones primitivas. Por ejemplo, la función factorial.
- Una expresión principal para ser evaluada (entendido como aplicación de una de las funciones definidas sobre otros datos). Por ejemplo: fact(2*fact(2)).
- Patrones de listas, como lista vacía [], y la lista no vacía (x:xs), patrón sucesor de los números naturales e instrucción *where* fueron

tomadas especialmente de Haskell. La instrucción *where* sólo es de un nivel.

En principio se iban a tratar los tipos predefinidos como son:

- Simples
  - Booleanos
  - Caracteres
  - Cadenas
  - Números
    - Reales
    - Naturales
- Tuplas
- Listas
- Arrays
- Árboles
  - Binarios
    - Con elementos en los nodos intermedios
    - Sin elementos en los nodos intermedios
  - 2 - 3
  - Generales

Al final se han implementado los siguientes tipos: booleanos, reales, naturales, tuplas y listas.

4.3. Limitaciones del lenguaje fuente

Como se ha comentado anteriormente aunque el lenguaje esta basado en ML (específicamente en Moscow ML), no se ha tomado toda la gramática del lenguaje sino un subconjunto de la misma.

Limitaciones del lenguaje fuente:

- Sin inferencia ni comprobación de tipos
- Sin listas intensionales, que se podrían incluir más adelante
- Entre los tipos predefinidos no se han incluido los caracteres, cadenas y arrays por simplicidad.
- No hay guardas, la instanciación se hace sólo por patrones, no por casos.
- Sin módulos
- Sin tipos definidos por el programador
- Sin orden superior, por consiguiente, sin λ-abstracción ni secciones

4.4. Gramática del lenguaje fuente

Después de varias revisiones de la gramática del lenguaje ML y Haskell se ha elegido varias características de los dos para nuestro lenguaje funcional, se ha llegado a la especificación de la gramática recogida a continuación. Podremos ver la gramática más desarrollada y su implementación en JAVACC en el Apéndice A.

Declaraciones

```
ldec::= dec ; [ldec]            lista de declaraciones
dec::= fun id latpat = exp      declaración de una función
latpat::= atpat [latpat]        lista de patrones
```

Patrones

```
atpat::= _                      patrón universal
     |cte                       constante
     |id                        variable (literal)
     |()                        tupla vacía
     |(x:xs)                    patrón lista no vacía   x,xs = var o _
     |[]                        patrón lista vacía
     |(x + cte + ... + cte)     patrón de naturales
     |(pat1, … , patn)          tupla no vacía              n>=1
     |[pat1, … , patn]          patrón compuesto de lista   n>=1

pat::= var atpat                valor construido
       |atpat                   un patrón atómico

patWhere::= id                  una variables
          |(id1, … , idn)       tupla de variables
```

Expresiones

```
exp::= if exp then exp else exp    sentencia if then else
       |case exp of match          sentencia case
       |exp op exp                 expresiones con operadores
                                   aritmético-lógicos
       |where patWhere = exp       claúsula where
       |infexp



infexp::= appexp infexp            aplicación
          |appexp
appexp::= cte                      constante
          |var                     variable (literal)
          |()                      tupla vacía
          |(exp1, ... , expn)      tupla no vacía de expresiones      n>=1
```

```
          |(exp:exp)                  lista no vacía de expresiones
          |[]                         lista vacía
          |[exp1, ... , expn]         lista de expresiones                n>=1
match::= mrule <| match>              lista de guardas
mrule::= pat => exp                   una guarda
```

5. LENGUAJE DE TRANSFORMACIÓN

5.1. Descripción del lenguaje de transformación

El lenguaje de transformación es mucho más simple que el lenguaje fuente. Básicamente es un lenguaje de macros, que consiste en el nombre de la regla que se quiere aplicar seguido de una serie de argumentos dependiendo dicha regla. Para aplicar las diferentes reglas de transformación se han utilizado ecuaciones y ajuste de patrones.

5.2. Características del lenguaje de transformación

Se va a usar un lenguaje de transformación sencillo. El entorno de transformación se quiere hacer cómodo para el usuario, visualmente atractivo y fácil de usar. Se va a incorporar un sistema de ayuda para el usuario. Las principales instrucciones de transformación son:

- Definición
- Instanciación
- Pliegue
- Despliegue
- Abstracción
- Leyes

5.3. Limitaciones del lenguaje de transformación

El lenguaje de transformación tiene algunas limitaciones.

- Al no haber comprobación ni inferencia de tipos, el usuario debe suministrar esta información en las transformaciones. Por esto, la especificación de varias reglas deben tener más parámetros de los podrían tener si el sistema contara con alguna de las partes antes mencionadas. Debido a esta causa es el usuario el que debe determinar algunos aspectos de la aplicación de reglas.

- Si un usuario quiere instanciar un parámetro de una función debe indicarle el tipo del parámetro debido a que el sistema no determina el tipo.

- El usuario siempre tiene que indicar en qué línea quiere aplicar la regla de transformación, no es suficiente con el nombre de la función. En caso de pliegue y despliegue también tiene que indicar

la línea donde se encuentra la función que se utiliza para hacer el pliegue o despliegue.

5.4. Formato y descripción detallada de las reglas de transformación

Aquí se especifican detalladamente las diferentes funciones que tiene nuestro lenguaje de transformación, y la manera en que pueden ser utilizadas cada una de ellas. Aunque en este apartado se ofrece una descripción formal, en el **apéndice B** se muestra una colección de ejemplos del lenguaje de transformación aplicado a casos muy variados con diferentes encajes de patrones.

5.4.1. Reglas de transformación

Primero se describen las instrucciones del sistema de pliegue despliegue y después describimos otras instrucciones del sistema.

5.4.1.1. Definición

Formato de la definición:

Instrucción	Nombre	Argumentos
Define	**define, definir, def**	<fun>

-- fun – toda la función a definir que se ajusta a la gramática del lenguaje fuente.

Descripción de la definición:

Define una nueva función en el programa que se está transformando. La función que introduce el usuario debe ser sintácticamente correcta o se producirá un error.

5.4.1.2. Instanciación

Formato de la instanciación:

Instrucción	Nombre	Argumentos
instance	**instance, instanciar, ins**	<numLin> <tipo> <iden>

-- numLin – línea de la función a instanciar.

-- tipo – tipo de la variable que se va a instanciar. El tipo puede ser ***int*** para instanciar enteros o ***list*** para las listas.

-- iden – el nombre del identificador de la función que se va a instanciar.

Descripción de la instanciación:

Realiza un nuevo ejemplar de una función definida en el programa. Habrá dos casos para hacer la instanciación: el caso base y el caso inductivo. Si en <tipo> se declara un ***int***, la instanciación de cada identificador <iden> será 0 y (n+1), siendo n una variable libre. Si en tipo se declara ***list***, se instanciará la variable <iden> con [] y (x:xs), siendo x y xs variables libres.

Formato de la sustitución:

Instrucción	Nombre	Argumentos
sustitución	**sustitucion, sust, st**	<numLin> <iden> <valor>

-- numLin – línea de la función a sustituir.

-- iden – el nombre del identificador de la función que se va a sustituir.

-- valor – valor de sustitución que se va a realiazar.

Descripción de la sustitución:

Realiza una sustitución de una variable por una expresión dentro de una función definida en el programa.

5.4.1.3. Despliegue

Formato del despliegue:

Instrucción	Nombre	Argumentos
unfold	**unfold, despliegue, unf, des**	<numLin> <nomFunc> <numLinUnf>
		<numLin> <numApar> <nomFunc> <numLinUnf>

-- numLin – línea de la función a donde se quiere hacer despliegue de otra función del programa.

-- numApar – número de aparición de la función que queremos desplegar. Este argumento es opcional. Si no aparece la instrucción hace el despliegue en la primera aparición de la función a desplegar.

-- nomFunc – nombre de la función que se quiere desplegar.

-- numLinUnf – línea donde se encuentra la función con la que se va a realizar el despliegue.

Descripción del despliegue:

Sustituye una función en sentido hacia la derecha (->), es decir, se cambia el valor de la función con el parámetro que tiene por la parte derecha de su definición. La función <nomFunc> que se quiere desplegar debe estar previamente definida en la línea <numLinUnf>, en caso contrario la instrucción no hace nada.

Al hacer el despliegue se tienen en cuenta todos los posibles encajes de patrones (Ver apéndice).

5.4.1.4. Pliegue

Formato del pliegue:

Instrucción	Nombre	Argumentos
fold	**fold, pliegue**	<numLin> <funFold> <nomFunc> <numLinFold>
		<numLin> <funFold> <numApar> <nomFunc> <numLinFold>

-- numLin – línea de la función donde se quiere hacer el pliegue de otra función del programa.

-- funFold – expresión entrecomillada del trozo de la parte derecha de la función que queremos plegar.

-- numApar – número de aparición de la función que queremos plegar. Este argumento es opcional. Si no aparece la instrucción hace el despliegue en la primera aparición de la función a desplegar.

-- nomFunc – nombre de la función que se quiere plegar.

-- numLinFold – línea donde se encuentra la función con la que se va a realizar el pliegue.

Descripción del pliegue:

Sustituye una expresión en sentido hacia la izquierda (<-), es decir, se cambia el valor de la función con el parámetro que tiene por la parte izquierda de su definición.

Al hacer el pliegue se realiza encaje de patrones de los parámetros que tiene la función actual con la definición de la función de pliegue. Hay que asegurase de que la función que se quiere plegar está previamente definida y que la expresión tiene el mismo patrón que la parte derecha de la definición de la función.

5.4.1.5. Abstracción

Formato de la abstracción:

Instrucción	Nombre	Argumentos
abstract	**abstract, abs**	<numLin>
		<numLin> <parteAbs>
		<numLin> <parteAbs> <numApar>

-- numLin – línea de la función a donde se quiere hacer la abstracción.

-- parteAbs – expresión entrecomillada del trozo de la parte derecha de la función que queremos abstraer. Este argumento es opcional, si no se selecciona una parte a abstraer, por defecto, el programa tomará toda la parte derecha de una función.

-- numApar – número de aparición del la parte de que se quiere abstraer en caso de que se repita su aparición más de una vez. Este argumento es opcional. Si no se usa este argumento y hay más de una aparición de la parte que se quiere abstraer el programa por defecto seleccionará la primera aparición.

Descripción de la abstracción:

Realiza una sustitución local en una función de una expresión por un parámetro que el usuario introduce que tendrá el mismo valor que la expresión sustituida. Para hacer la abstracción se usa *where*.

La función obtenida será equivalente pero con una declaración local del parámetro elegido por el usuario.

5.4.1.6. Leyes

Formato de la definición de ley:

Instrucción	Nombre	Argumentos
define law	deflaw, defley, dl	<nombre> <ley>

-- nombre – nombre de la ley.

-- ley – definición de la ley que aplicaremos posteriormente a las transformaciones. La definición de la ley debe tener una forma especifica: ***exp <=> exp ,*** donde exp es una expresión genérica que encaja con la expresiones de la gramática y el símbolo <=> es la doble implicación de la ley, para posteriormente utilizar la ley en los dos sentidos.

Descripción de la definición de ley:

Introduce una ley que posteriormente podrá ser usada en las transformaciones. La ley definida se incorpora a nuestro conjunto de leyes. Por defecto en la aplicación hay un conjunto de leyes predefinidas, pero con esta instrucción cada usuario puede definir sus propias leyes para posteriormente aplicarlas.

Algunas de las leyes que están en el sistema son la asociatividad (+,*) entre expresiones, el elemento neutro, la aplicación de una expresión condicional, etc.

Formato del uso de la ley:

Instrucción	Nombre	Argumentos
use law	uselaw, usarley, ul	<nombre> <numLin>
		<nombre> <numLin> <numApar> <parteLey>
		<nombre> <numLin> <numApar> <sentido> <parteLey>

-- nombre – nombre de la ley.

-- numLin – línea de la función a donde se quiere aplicar la ley.

-- numApar – número de aparición del la parte a la que se quiere aplicar la ley en caso de que se repita su aparición más de una vez. Este argumento es opcional. Si no se usa este argumento y hay más de una aparición de la parte que se quiere abstraer el programa por defecto seleccionará la primera aparición.

-- sentido – este parámetro indica el sentido de la aplicación de la ley. Por lo general el sentido de la ley será de izquierda a derecha, pero el usuario puede querer aplicar una ley de derecha a izquierda. Este parámetro es opcional y puede ser ***ID*** (en caso de ser de izquierda a derecha) o ***DI*** (en caso de ser de derecha a izquierda).

-- parteLey – expresión entrecomillada del trozo de la parte derecha de la función a la cual queremos aplicar la ley. Este argumento es opcional, si no se selecciona, por defecto, el programa tomará toda la parte derecha de una función para aplicar la ley.

Descripción del uso de la ley:

Esta instrucción aplica una ley definida previamente a una función o a una parte de ella. Si solamente incluimos los argumentos del nombre y el número de línea, el sistema intentará aplicar la ley sobre toda la parte derecha de la función que está en la <numLin>. Si se le indica con el resto de parámetros dónde queremos aplicarla, el sistema la aplicará específicamente en esa parte.

En caso de que se esté intentando aplicar una ley que no encaje con la función a la cual se le quiere aplicar, la instrucción no hará nada.

Formato del constant folding:

Instrucción	Nombre	Argumentos
constant folding	**consfold, cf**	<numLin>
		<numLin> <parteCF>

-- numLin – línea de la función a donde se quiere aplicar el constant folding.

-- parteCF – expresión entrecomillada del trozo de la parte derecha de la función a la cual queremos aplicar el constant folding. Este argumento es opcional, si no se selecciona, por defecto, el programa tomará toda la parte derecha de una función para intentar reducirla.

Descripción del constant folding:

El constant folding sirve para reducir total o parcialmente expresiones compuestas por constantes a expresiones con las constantes simplificadas si es posible operar con ellas.

El constant folding se hace tanto de números como de booleanos. Se realiza el constant folding de las partes derecha e izquierda de la

declaración. Si en una expresión están mezclados números o booleanos con variables, el constant folding no se hace.

Los operadores && y || son perezosos, es decir, *false* && x = *false*. En el caso de un *if - then - else*, es también perezoso, es decir, primero se evalúa la condición y si se puede (si no hay variables en medio) entonces el resultado es la rama correspondiente. No se evalúa nunca la rama que no hace falta para evitar la no terminación.

En cuanto a los casos de error, el sistema comprueba si se ha producido una división por cero e informa al usuario en ese caso. También detecta si se ha realizado una operación entre números y booleanos y también informa del error.

5.4.2. Opciones propias del sistema

Estas opciones sirven para realizar algunas utilidades similares a las que hacen otros sistemas de transformación como MAG [MAG] para cargar y salvar *scripts*, deshacer instrucciones de transformación aplicadas anteriormente, etc.

Cargar

Formato de la opción cargar:

Instrucción	Nombre	Argumentos
load	**loadProgram, load, ld**	<nombre>

-- nombre – nombre del archivo sin extensiones.

Descripción de la opción cargar:

Carga un *script* con una lista de instrucciones de transformación. Este archivo se encuentra en la carpeta /*Scripts*. Si el archivo no existe la opción no hace nada.

Guardar

Formato de la opción guardar:

Instrucción	Nombre	Argumentos
save	**saveProgram, save, sv**	<nombre>

-- nombre – nombre del archivo sin extensiones.

Descripción de la opción guardar:

Salva un *script* con una lista de instrucciones de transformación. Este archivo se guarda en la carpeta */Scripts*.

Deshacer

Formato de la opción deshacer:

Instrucción	Nombre	Argumentos
undo	undo, deshacer, dh	<numIns>

-- numIns – número de instrucción de transformación hasta la cual queremos deshacer.

Descripción de la opción deshacer:

Esta opción permite deshacer varias instrucciones de transformación que el usuario desee rectificar.

Copiar

Formato de la opción copiar:

Instrucción	Nombre	Argumentos
copy	copiar, copy, cpy	<Nombrefuente> <numIns>

-- Nombrefuente – indica donde se encuentra la instrucción que se quiere copiar (t1 o lib).

-- numIns – posición de la instrucción que se quiere copiar.

Descripción de la opción copiar:

Esta opción permite copiar instrucciones que se encuentra en un determinado panel al panel de trabajo. Los paneles pueden ser el panel donde se guardan las instrucciones antes de instanciarse y la librería de funciones.

6. TÁCTICAS DE TRANSFORMACIÓN

Hay diversas técnicas de transformación de programas. Estás técnicas se basan en el uso de diversas tácticas algebraicas para poder llegar a obtener el programa final transformado. Las principales tácticas son la especialización, la fusión, la generalización, la evaluación parcial, la deforestación, la recursión final, el tuplamiento, la programación dinámica, la promoción de filtros, el backtracking y la paralelización [DSMT].

A continuación se muestra el mecanismo de cada una de ellas.

6.1. Paso de recursión no final a recursión final

En las funciones recursivas lineales (o sea, con una llamada recursiva), el caso recurrente precisa en general posterior procesamiento. Esta táctica precisa eliminar dicho proceso posterior transformando esas funciones en funciones con recursividad final donde la llamada recursiva subsidiaria representa directamente al resultado sin posterior tratamiento.

Una solución recursiva se basa en un análisis de los datos, para distinguir los casos de solución directa y los casos de solución recursiva. En la recursión simple (lineal) una acción recursiva tiene recursión simple si cada caso recursivo realiza exactamente una llamada recursiva.

La recursión final es un caso particular de recursión simple donde la función de combinación se limita a transmitir el resultado de la llamada recursiva. Se llama final porque lo último que se hace en cada pasada es la llamada recursiva. El resultado será siempre el obtenido en uno de los casos base.

Ejemplo programa inicial:

```
fun f x = if triv x then base x else  a x + f (t x);
```

Programa transformado:

```
fun f x = if triv x then base x else a x + f ( t x ) ;
fun g ac x = if triv x then ac + ( base x ) else g ( ac + a x ) ( t x ) ;
fun g ac x = ac + f x ;
fun f x = g z x ;
```

6.2. Especialización y evaluación parcial

El objetivo básico de la evaluación parcial consiste en, dado un programa y **parte** de sus datos de entrada, generar una versión especializada del programa original para dichos datos de entrada (por ello, a menudo se denominan también técnicas de **especialización**). Por supuesto, se espera que el programa especializado compute los mismos resultados que el programa original si se fijan los datos de entrada empleados para realizar la evaluación parcial. Es decir, si tenemos un programa **P** con dos parámetros, **in1** e **in2**, y realizamos la evaluación parcial de **P** con respecto a **in1**, obteniendo el programa especializado **P_in1**, cabe esperar que **P(in1,in2)** compute los mismos resultados que **P_in(in2)**.

Las técnicas de evaluación parcial se han aplicado a diferentes lenguajes de programación (Prolog, Haskell, Lisp, Scheme, C, Java, etc) y con diferentes propósitos: mejorar la eficiencia de programas (o procedimientos) de propósito general, optimizar bases de datos, mejorar la eficiencia de los algoritmos de cálculo científico, generar compiladores a partir de intérpretes de forma automática, etc.

Ejemplo programa inicial:

```
fun exp a 0 = 1;
fun exp a (n+1) = a exp a n;
```

Programa transformado:

```
fun exp 2 0 = 1 ;
fun exp 2 ( n + 1 ) = 2 exp 2 n ;
```

6.3. Fusión

La fusión une dos piezas de código, digamos un productor y un consumidor, dando lugar a una sola que ahorra cómputos y el consumo de memoria intermedios.

Ejemplo programa inicial:

```
fun media l = sum l / long l ;
fun sum [ ] = 0 ;
fun sum ( x : xs ) = x + sum xs ;
fun long [ ] = 0 ;
fun long ( x : xs ) = 1 + long xs ;
```

Programa transformado:

```
fun media l = n / o
               where (n,o)= sumlong l ;
fun sum [ ] = 0;
fun sum ( x : xs ) = x + sum xs ;
```

```
fun long [ ] = 0 ;
fun long ( x : xs ) = 1 + long xs;
fun sumlong [ ] = ( 0, 0) ;
fun sumlong (o:os)=((o+n),(1+p) )
          where ( n , p ) = sumlong os;
(*)fun sumlong xs =(sum xs,long xs ) ;
```

6.4. Deforestación

La deforestación es un caso particular de la táctica de fusión. Se utiliza cuando hay funciones que producen tipos de datos algebraicos, y otras funciones que los consumen. La deforestación elimina la creación y el consumo de estos constructores. Los tipos de datos más típicos a los que se aplica la deforestación son las listas, los árboles; de ahí el nombre de deforestación) y también podrían ser los naturales, se consideran a menudo como un tipo de datos algebraico, con los constructores Cero y Sucesor, que se podrían considerar como el sucesor de un número.

Ejemplo programa inicial:

```
fun f a b = suma (inter a b);
fun inter a b = if (a>b) then [] else (a: inter (a+1) b);
fun suma [] = 0;
fun suma (x:xs) = x + suma xs;
```

Programa transformado:

```
fun f a b = (if ( a > b ) then 0 else a + f ( a + 1 ) b ) ;
fun inter a b = if ( a > b ) then [ ] else ( a : inter ( a + 1 ) b ) ;
fun suma [ ] = 0 ;
fun suma ( x : xs ) = x + suma xs ;
fun f a b = suma ( inter a b ) ;
```

6.5. Tuplamiento

El tuplamiento es una táctica de generalización de resultados. La estrategia de tuplamiento puede ser usada para la fusión horizontal de bucles por combinación de llamadas recursivas y también para eliminar las llamadas redundantes para una clase de programas. Podemos aplicar tuplamiento sobre funciones con llamadas recursivas múltiples y argumentos parámetros acumulativos sin riesgo de no terminación.

Ejemplo programa inicial:

```
fun fib 0 = 1;
fun fib 1 = 1;
fun fib (n+2) = fib n + fib (n+1);
fun fibPar n= (fib n, fib (n+1));
```

Programa transformado:

```
fun fib 0 = 1 ;
fun fib 1 = 1 ;
fun fib ( n + 2 ) = fib n + fib ( n + 1 ) ;
fun fibPar 0 = ( 1 , 1 ) ;
fun fibPar ( o + 1 ) = ( p , n + p )
          where ( n , p ) = fibPar o ;
fun fibPar n = ( fib n , fib ( n + 1 ) ) ;
```

6.6. Programación dinámica

La programación dinámica es un algoritmo importante de diseño técnico. La Programación Dinámica tiene sentido aplicarla por razones de eficiencia, porque presenta un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas y ha fracasado. Donde tiene mayor aplicación es en la resolución de problemas de optimización. En este tipo de problemas se pueden presentar distintas soluciones, cada una con un valor, y lo que se desea es encontrar la solución de valor óptimo (máximo o mínimo).

Por el momento no se puede aplicar en el sistema por las limitaciones existentes (no se han implementado los arrays).

6.7. Promoción de filtros

La promoción de filtros altera el momento en que se realizan operaciones de test, suprimiendo cómputos innecesarios.

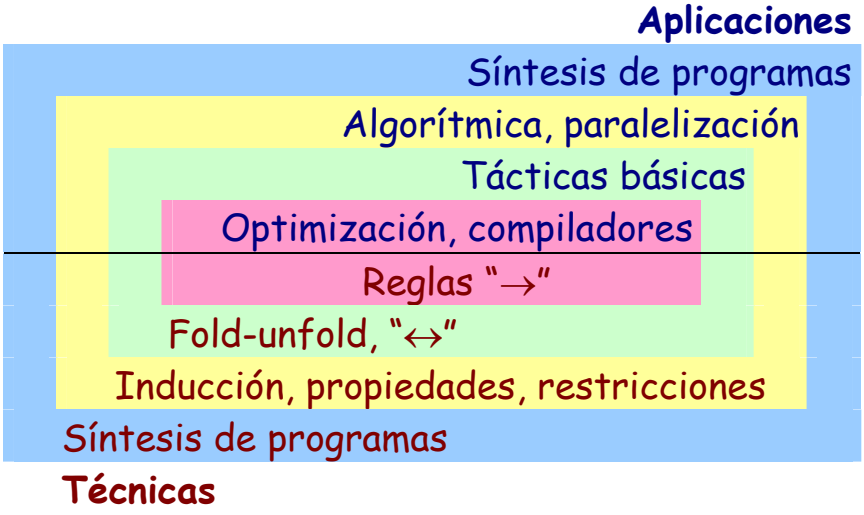
```
filter p . map f = map f . filter (p . f)
```

Esta táctica tampoco se puede aplicar en el sistema porque no hay orden superior.

6.8. Backtracking

Backtracking es el método de resolución de problemas según el cual se hace una búsqueda sistemática de una o todas las soluciones del problema por medio de repetidamente intentar extender una solución aproximada de todas las formas posibles. Si en algún momento resulta que la solución es falsa retrocede al último punto de elección donde todavía había alternativas de encontrar la solución.

Esta táctica no se puede aplicar en el sistema por razones similares que la táctica anterior.


Aplicaciones
Síntesis de programas
Algorítmica, paralelización
Tácticas básicas
Optimización, compiladores
Reglas "→"
Fold-unfold, "↔"
Inducción, propiedades, restricciones
Síntesis de programas
Técnicas

7. IMPLEMENTACIÓN DEL SISTEMA

7.1. Requisitos a nivel de implementación

El lenguaje de implementación que se va a utilizar para hacer las transformaciones es Java, sencillamente porque para las transformaciones se necesita recorrer el árbol sintáctico y este árbol es el que proporciona el parser que está hecho con JavaCC, quiere decir que su código está completamente en Java.

Para poder realizar todas las transformaciones sobre el árbol sintáctico hay que tener estructuras que sean eficientes y manejables para programar las transformaciones de la manera más sencilla, funcional y eficiente.

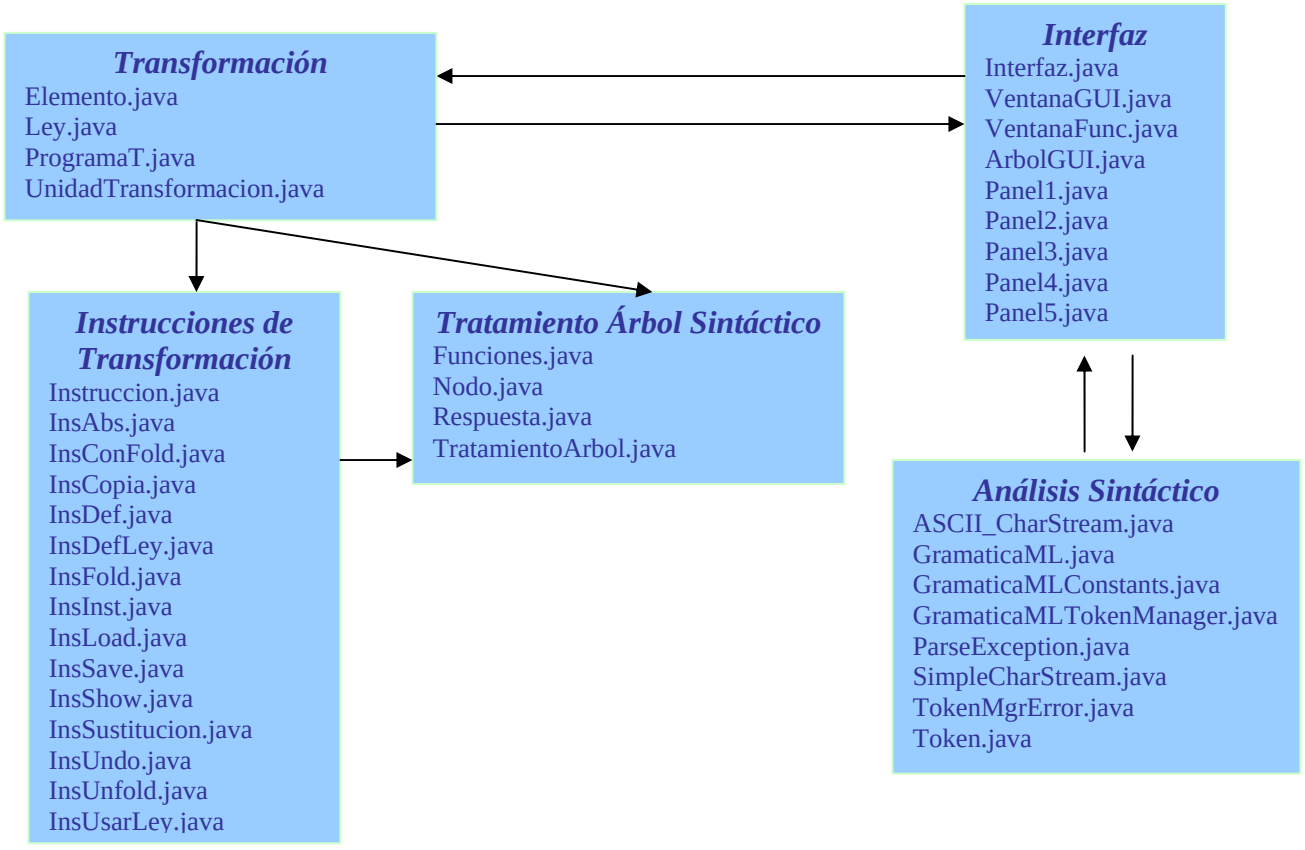
La estructura que se ha elegido es la de un árbol, y la implementación de las transformaciones complejas se ha hecho mediante el recorrido en preorden o postorden de ese árbol. A priori se necesita que cada uno de los nodos tenga un identificador de su nodo, un identificador del nodo padre, una lista de hijos, un identificador que indique si es nodo hoja o no. Esta estructura es muy útil para realizar la regla de constant folding, ya que necesitamos saber la prioridad de los operandos y esta estructura mantiene implícita dicha propiedad.

El sistema está implementado con JAVA y consta de varias partes, en primer lugar un módulo para el parser del lenguaje fuente, que está implementado con la herramienta JAVACC, después un módulo que incluye todas las reglas necesarias para transformar la estructura de datos del árbol sintáctico. Posteriormente hay otro módulo que se encarga de manejar todas las instrucciones de transformación, y finalmente se encuentra en interfaz, donde se conectan todos los módulos y crea el entorno visual que manejaran los usuarios.

7.2. Descripción de la arquitectura

La arquitectura del sistema consta de 5 partes que se conectan entre sí, y a su vez cada parte está compuesta de varias clases. El diseño se ha hecho lo más modular posible para que el sistema sea fácilmente ampliable y modificable.

En el siguiente esquema se puede observar qué clases componen cada módulo y cómo se interconexionan.



7.2.1. Módulo de análisis sintáctico

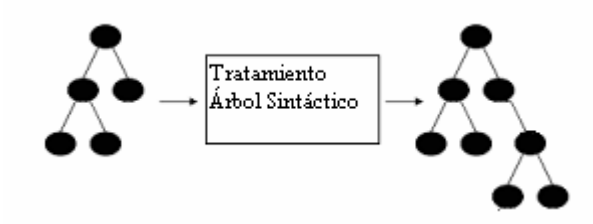
El módulo de análisis sintáctico se encarga de generar la estructura del árbol sintáctico de nuestro lenguaje fuente. Dicha estructura de árbol está implementada con nodos que se enlazan entre sí formando un árbol.

Cada nodo tiene información sobre su contenido, si es un no terminal, el nombre de la producción y si es un terminal, el nombre del token. Además cada nodo tiene una lista de hijos con enlaces a los siguientes nodos y los métodos necesarios para acceder y modificar el árbol.

Este módulo está generado con la herramienta JAVACC. En JAVACC editamos un archivo con extensión .jj que contiene la gramática de ML. Dicha gramática debe estar acondicionada a derechas ya que JAVACC es una herramienta para generar traductores predictivos recursivos y no admite recursividad a izquierdas.

7.2.2. Módulo de tratamiento del árbol sintáctico

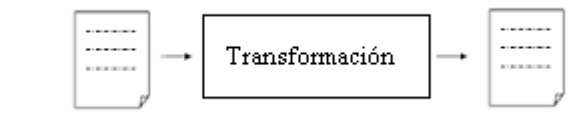
En este módulo tenemos una clase TratamientoArbol donde se realizan la mayoría de operaciones con árboles. En esta clase y en otras, usamos varias funciones para transformar el árbol. Sintácticamente un árbol está compuesto por una o varias declaraciones. En esta clase consideramos cada declaración como un subárbol. Sobre cada subárbol se podrán realizar distintas operaciones, como obtener las variables de la ecuación, sustituirlas o sustituir una parte de la declaración por otra nueva, concatenar subárboles, eliminarlos, etc.



7.2.3. Módulo de transformación

En este módulo se implementa un manejador de instrucciones de transformación. Cada instrucción de transformación está formada por el nombre de la propia instrucción más los parámetros de cada instrucción.

Debido a la implementación que se ha realizado, para usar un mismo tipo de instrucción se puede usar distintos nombres, uno largo y otro abreviado o en inglés, etc. Por esto se puede ampliar fácilmente el repertorio de instrucciones con respecto a las que ya hay. Además una misma función puede usarse con diferente número de parámetros si se quiere aplicar a una parte de una ecuación o a toda.



A todas las instrucciones de transformación hay que indicarles el número de la línea del programa que queremos transformar y alguna otra información.

La clase Instrucción es una clase genérica de la cual extienden el resto de clases que son instrucciones. Desde estas clases se hacen muchas referencias a funciones del módulo Tratamiento Árbol Sintáctico.

Posteriormente todas las instrucciones de transformación aplicadas a un programa se almacenan para poder ver toda la manipulación del programa desde el principio e incluso poder deshacer cambios que se hayan tomado.

7.2.4. Interfaz

En el interfaz de usuario es donde se conecta el funcionamiento de todas las partes del mismo y el usuario puede transformar los programas funcionales.

El entorno de usuario es una ventana que cuenta con un menú contextual y unos botones de acceso rápido en la parte superior, varias pestañas para repartir el espacio de la aplicación en el centro de la ventana, y la consola en la parte inferior para transformar el programa mediante instrucciones de transformación.

El menú contextual cuenta con opciones de Archivo (subopciones: Nuevo, Cargar Programa, Salvar, Fijar programa de partida, Salir), Herramientas (subopciones: Ver Árbol Derivación, Ver Árbol Derivación 2) y Ayuda (subopciones: Sobre las Transformaciones, Documentación del Proyecto, Acerca de…).

Los botones de acceso rápido son los de Nuevo, Cargar y Ejemplos de Instrucciones de Transformación.

Las pestañas son: Programa de Partida, Transformaciones, Leyes, Librerías y Programa Transformado.

En la pestaña Programa de Partida hay que introducir el programa fuente y pasarlo por el parser. En esta etapa se comprueba que el programa de origen sea sintácticamente correcto. Las leyes (pestaña Leyes) se pueden ampliar en cualquier momento por la consola. En la pestaña Programa Transformado se muestra el programa transformado con una "impresión bonita" y se muestran todas las instrucciones de transformación aplicadas al programa inicial.

8. HERRAMIENTAS DE DESARROLLO

Una vez decidido el lenguaje de implementación hay que elegir las herramientas auxiliares que se van a utilizar, o decidir que el parser se va a implementar a mano en Java. Esa última opción se consideró sólo por la razón de que la elección correcta y el aprendizaje de nuevas herramientas pueden llevar mucho tiempo. Se descartó debido a que la implementación manual del parser complicaría mucho la aplicación, además de lo complicado que puede llegar a ser ir corrigiendo detalles de la gramática, en caso de necesitarlo, si se implementa manualmente.

Para la implementación de este sistema se consideraron varias herramientas que podrían ser de gran ayuda en su implementación como por ejemplo:

JavaCC

JavaCC (Java Compiler Compiler) es el generador de párser más popular que se usa con aplicaciones Java. Un generador de párser es una herramienta que lee una especificación de gramática y la convierte en un programa Java que pueda reconocer la gramática.

Además del generador de párser en sí, JavaCC proporciona otras herramientas estándar relacionados a la generación de párser así como construcción de árboles (por medio de la herramienta JJTree incluido con JavaCC), acciones, depurador, etc.

LEX

LEX es una herramienta de los sistemas UNIX/Linux que nos va a permitir generar código C que luego podremos compilar y enlazar con nuestro programa. La principal característica de Lex es que nos va a permitir asociar acciones descritas en C, a la localización de las Expresiones Regulares que le hayamos definido. Para ello Lex se apoya en una plantilla que recibe como parámetro, y que deberemos diseñar con cuidado.

YACC

YACC (Yet Another Compiler-Compiler) es una herramienta que toma una especificación de gramática y escribe un analizador sintáctico en C que reconoce sentencias válidas para dicha gramática. Sirve para construcción de analizadores sintácticos de lenguajes del tipo LR(1) (LALR). Yacc emergió de los laboratorios Bell a mediados de los años setenta, y es indiscutiblemente la herramienta de análisis sintáctico disponible más exitosa. De ella se han producido numerosas variantes, como Berkeley Yacc y el bisonte de la fundación de software libre.

8.1. Elección de la herramienta

Como primera opción nos decantamos por utilizar LEX y el YACC ya que los tres ya lo conocíamos y estábamos familiarizados con estas herramientas. Al final hemos decido implementar nuestro compilador en JavaCC por varias razones, porque se puede integrar fácilmente con el lenguaje Java que es el lenguaje que hemos elegido para la implementación general del proyecto y además porque es muy fácil implementar en JavaCC una vez aprendida esta herramienta.
Para la implementación de nuestro lenguaje en JavaCC, se ha tenido que modificar varias cosas de la gramática sin cambiar la semántica propia del lenguaje, por ejemplo, la recursión a izquierdas que es una de las cosas que JavaCC no permite para su implementación. Otra de las cosas son las producciones lambda, que tampoco se permiten.

8.2. Características de JavaCC

JavaCC es una herramienta creada por Sun Microsystems. Sigue el estilo de herramientas independientes, cuya interacción es por la línea de comandos.

Usa expresiones regulares para el análisis lexicográfico y gramáticas LL(k) L-atributadas para el análisis sintáctico. Las gramáticas incluyen la operación de clausura y la estructura opcional, pudiéndose insertar las acciones semánticas en cualquier lugar de la producción. Para el reconocimiento usa "lookahead" local, que puede ser combinado con "lookahead" sintáctico y "lookahead" semántico. En la implementación se usa el método recursivo descendente, generándose una clase de Java la cual contendrá un método por cada no-terminal definido en la gramática.

Las acciones semánticas pueden escribirse en Java o cualquier extensión a este lenguaje, ya que el generador sólo chequea los delimitadores de las acciones semánticas. No hay necesidad de separar la especificación del analizador lexicográfico del sintáctico, aunque de estar presentes en el fichero, puede generarse código sólo para una de ellas si así se desea. La especificación es monolítica, y aunque se prometía la capacidad de dividir una especificación en varios ficheros en la documentación de versión 0.8pre1, en la versión 1.0 aún no se había incluido este recurso, para más, la documentación sigue siendo la misma que aparece en la versión 0.8pre1.

Si se desea depurar errores en el analizador, en el momento de generarlo esto debe ser especificado. Esto hace que se imprima una traza de la ejecución, que puede pedirse para el analizador lexicográfico o el sintáctico de forma no exclusiva. En el caso del analizador sintáctico, puede pedirse que se incluyan en la traza las acciones de las operaciones de "lookahead".

JavaCC por defecto chequea la existencia de recursividad izquierda, ambigüedad con un "lookahead" limitado, uso de expansiones en la cadena vacía, entre otros. Opcionalmente puede pedirse que se chequee la ambigüedad, pasando por alto las especificaciones de "lookahead" de la gramática.

ANÁLISIS DESCENDENTE o TOP-DOWN

JavaCC genera parsers recursivos descendentes. Si bien se elimina la posibilidad de utilizar recursión a izquierda, se permite la utilización de gramáticas más generales, la facilidad para depurar el código generado, la habilidad de parsear cualquier no terminal de la gramática, y la capacidad de pasar atributos en el árbol sintáctico durante el parsing.

ESPECIFICACIÓN LÉXICA Y GRAMATICAL EN UN ÚNICO ARCHIVO

Tanto las especificaciones léxicas como las gramaticales (expresiones regulares y strings, y producciones BNF, respectivamente) se escriben en el mismo archivo. De esta manera la gramática puede ser leída y mantenida más fácilmente gracias al uso de las expresiones regulares dentro de la gramática.

PERMITE EXTENDER ESPECIFICACIONES BNF

JavaCC permite extender especificaciones BNF en las especificaciones léxicas y gramaticales, mediante la utilización de expresiones regulares, tales como (A)*, (A)+. Las BNF extendidas descubren la necesidad de utilizar recursión a izquierda para algunas extensiones.

ESTADOS LÉXICOS Y ACCIONES LÉXICAS

JavaCC ofrece estados léxicos y la capacidad de agregar acciones léxicas, además de los conceptos de token, more, skip, cambio de estados, etc. Ello permite trabajar con especificaciones más claras, a la vez que permite mejor manejo de mensajes de error y warning de JavaCC.

ESPECIFICACIONES SINTÁCTICAS Y SEMÁNTICAS DE LOOKAHEAD

JavaCC genera por defecto un parser LL(1). sin embargo, puede haber porciones de la gramática que no son LL(1). JavaCC ofrece la posibilidad de resolver las ambigüedades shift-shift localmente al punto del conflicto. En otras palabras, el parser se vuelve LL(k) sólo en tales puntos pero se conserva LL(1) en el resto de las producciones, para obtener una mejor performance. Los conflictos shift-reduce y reduce-reduce no son objetivos de los parsers descendentes.

ANÁLISIS LÉXICO CASE-INSENSITIVE

Las especificaciones léxicas pueden definir tokens de manera tal que a nivel global no se diferencien las mayúsculas de las minúsculas en la especificación léxica completa, o en una especificación léxica individual.

CAPACIDADES EXTENSIVAS DE DEPURACIÓN

Por medio de la utilización de las opciones DEBUG_PARSER, DEBUG_LOOKAHEAD, y DEBUG_TOKEN_MANAGER, el análisis del parsing y los pasos de procesamiento de tokens pueden realizarse en profundidad.

PREPROCESADOR PARA DESARROLLO DE ÁRBOLES

JavaCC incluye JJTree, un preprocesador para el desarrollo de árboles, con características muy poderosas.

MUY BUEN REPORTE DE ERRORES

De entre los generadores de parsers, JavaCC se halla entre los que tienen mejor manejo del reporte de errores. Los parsers generados por JavaCC son capaces de localizar exactamente la ubicación de los errores, proporcionando información diagnóstica completa.

GENERACIÓN DE DOCUMENTACIÓN

JavaCC incluye una herramienta llamada JJDoc que convierte los archivos de la gramática en archivos de documentación.

INTERNACIONALIZACIÓN

El analizador léxico de JavaCC puede manejar entradas Unicode, y las especificaiones léxicas también pueden incluir cualquier caracter Unicode. Esto facilita la descripción de los elementos del lenguaje, tales como los identificadores Java que permiten ciertos caracteres Unicode que no son ASCII, pero no otros.

AMPLIO USO DE LA COMUNIDAD

JavaCC es quizá el generador de parsers usado con aplicaciones Java más popular.

100 % JAVA

JavaCC corre en una plataforma completamente Java. Ello permite que pueda ser usado en diferentes máquinas sin problemas de portabilidad

9.

ESTADO DEL ARTE (Otros sistemas de transformación)

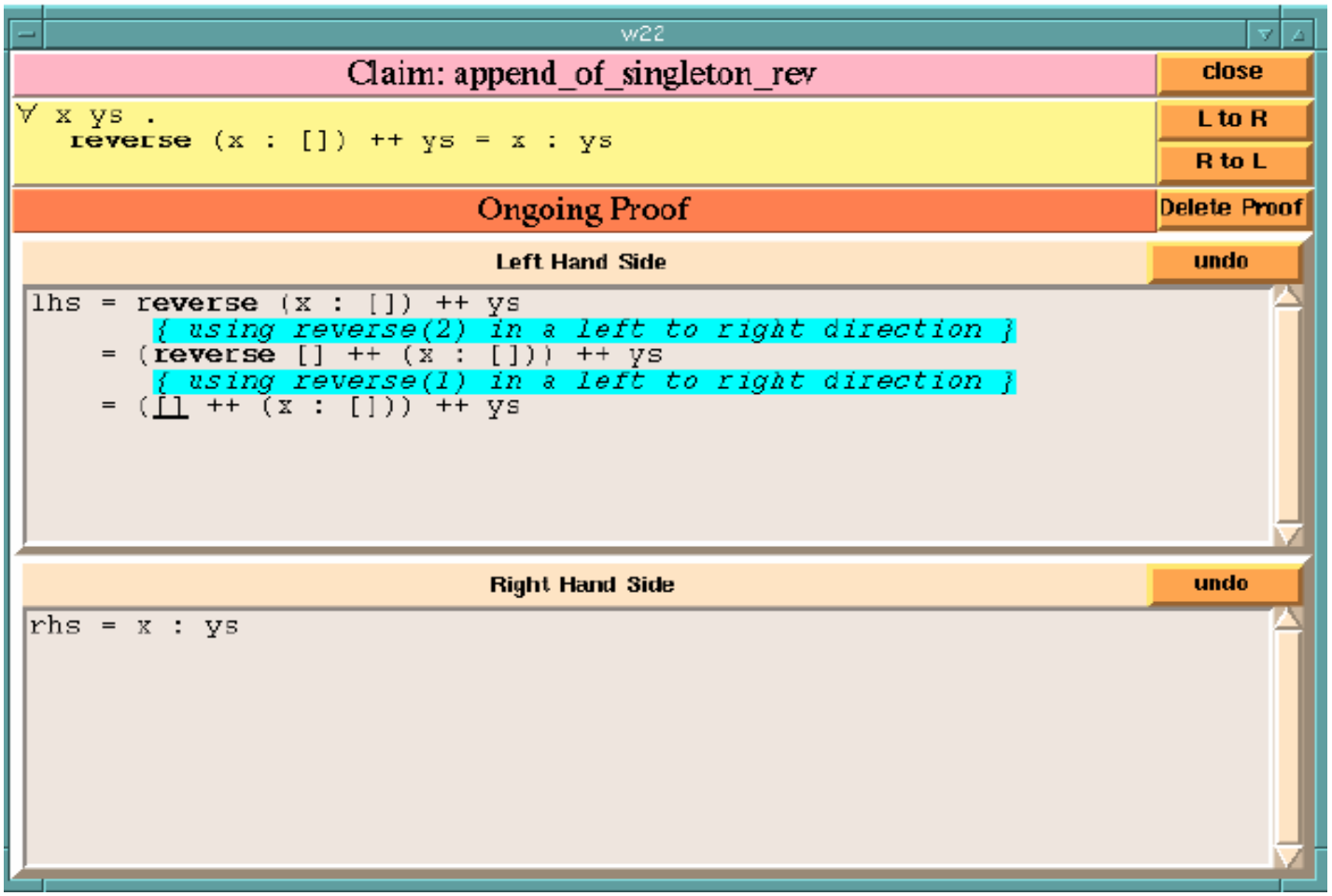
9.1. Introducción

Actualmente hay varios grupos de programadores en diferentes lugares del mundo donde se está trabajando e investigando con la programación funcional, uno de los más importantes es el grupo de programación funcional de la Universidad de York (York Functional Programming Group) [YFPG]. Uno de los proyectos de este grupo fue STARSHIP.

9.2. Referencias de otras aplicaciones de transformación

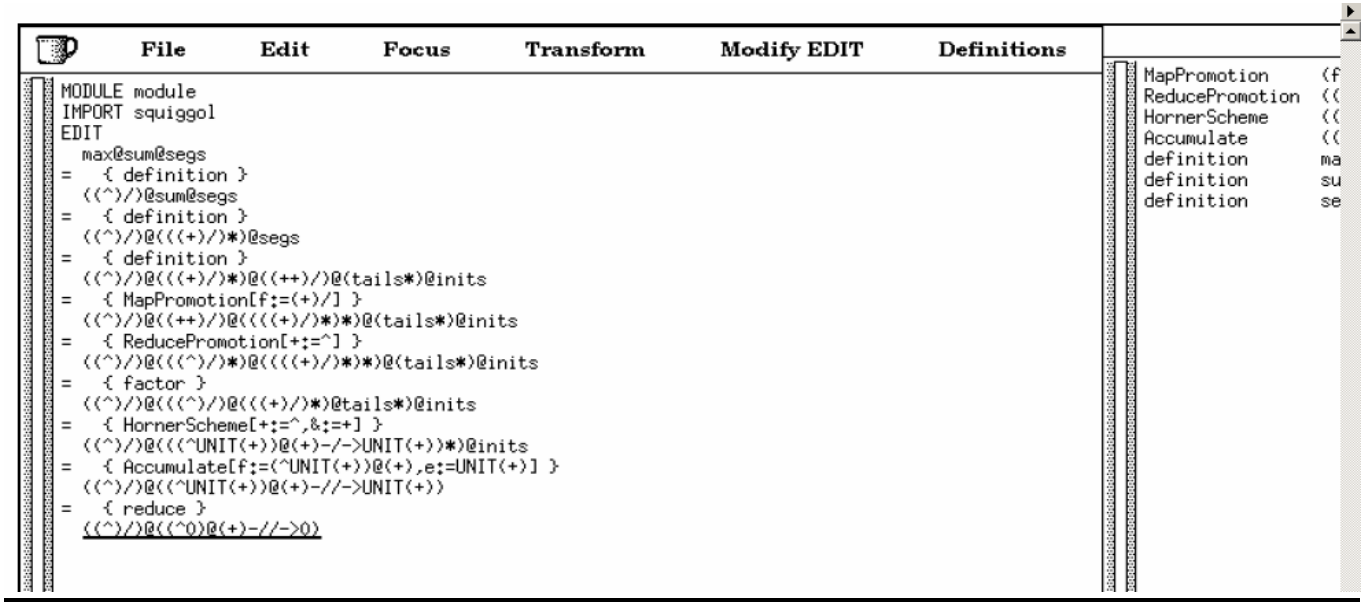
Sistema HERA (Haskel Equational Reasoning Asistant)

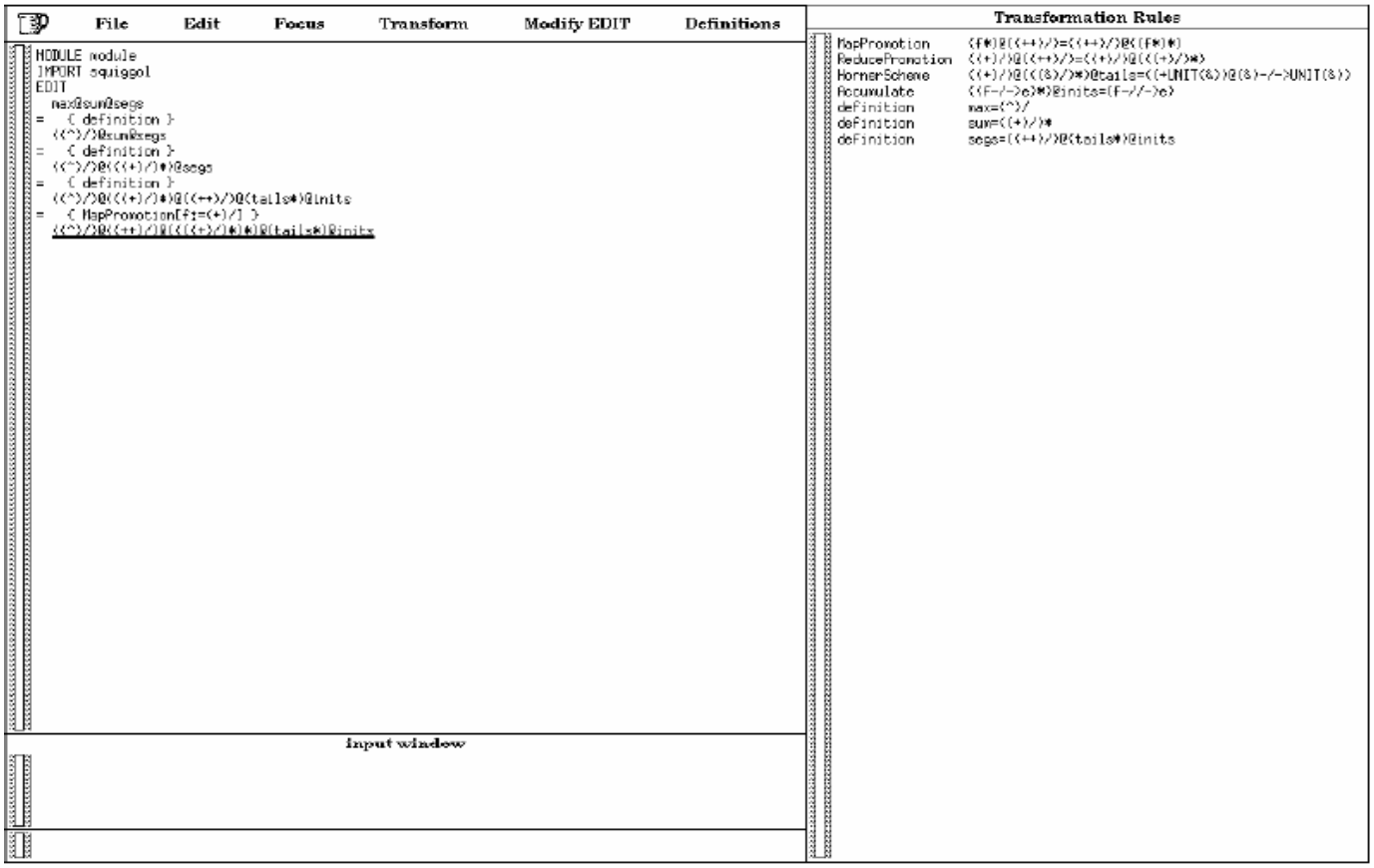
HERA es una aplicación escrita en Haskell que ayuda a los usuarios a construir y presentar pruebas del estilo del razonamiento mediante ecuaciones. Este proyecto fue desarrollado por Andy Gil para el departamnto de Ciencias de la Computación de la Universidad de Glasgow [HERA].



Sistema PROXAC

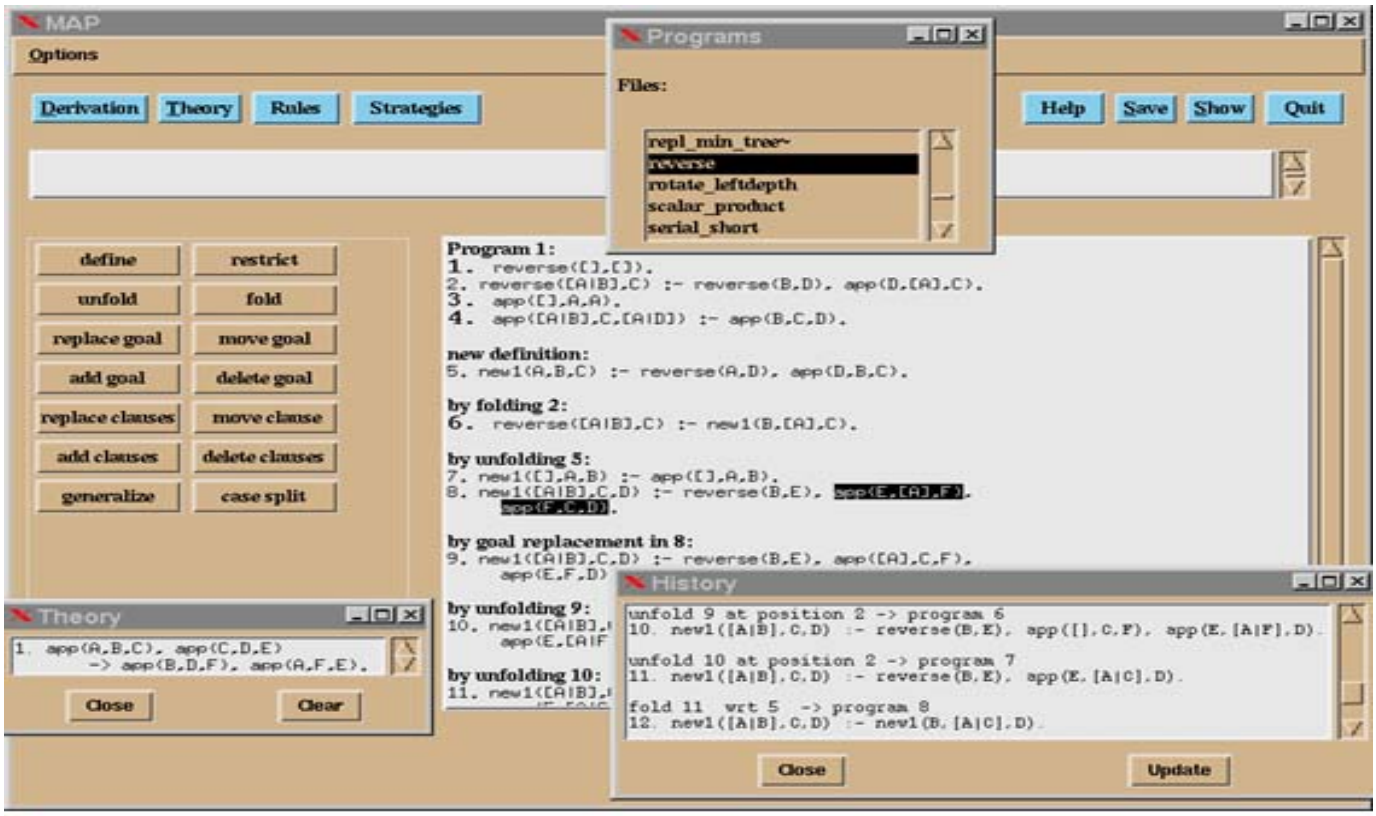
Como ya se ha mencionado anteriormente, este sistema trabaja con la notación de Bird-Meertens. Este sistema puede aplicar técnicas como la promoción de mapas y filtros y se pueden definir reglas para aplicarlas sobre funciones.





Sistema MAP

Este sistema de transformación es capaz de aplicar todas reglas del sistema de pliegue/despliegue. Una peculiaridad de este sistema es que trabaja con programación lógica y no con programación funcional. Utiliza PROLOG como lenguaje fuente para realizar las transformaciones. La aplicación fue desarrollada por la universidad de Roma [MAP].



Sistema STARSHIP

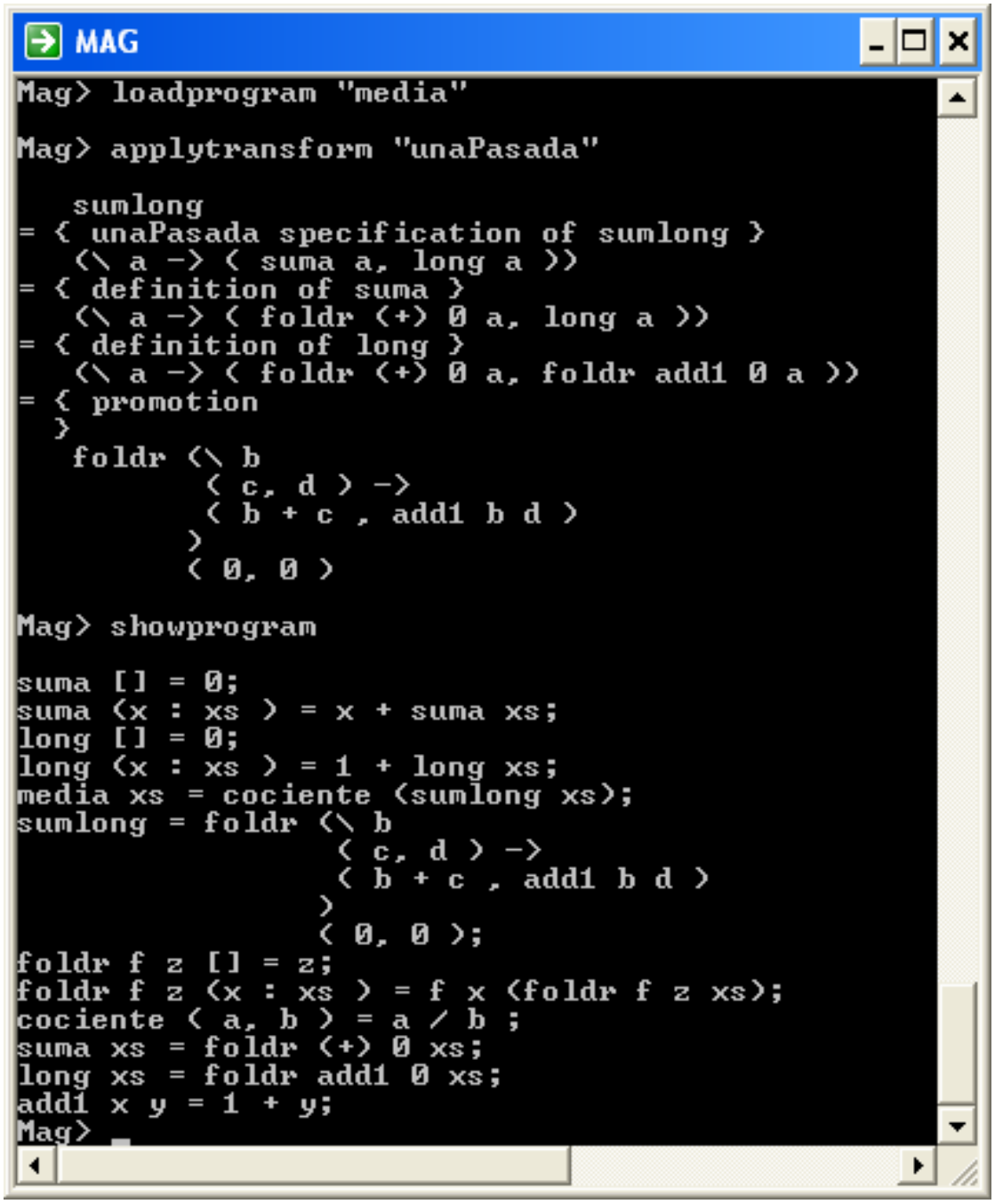
Sistema de transformación que permite el uso de las reglas del sistema de pliegue/despliegue, así como la definición y aplicación de leyes en el programa. Fue desarrollado por un grupo de programación funcional de la Universidad de York (York Functional Programming Group) [YFPG]. STARSHIP permite demostraciones por inducción simples en un lenguaje polimórfico y perezoso. Admite un amplio abanico de ejemplos, incluso con operaciones de entrada/salida. Si alguna suposición se vuelve falsa, el sistema automáticamente deshace los pasos que dependen de esa suposición. También se pueden almacenar, manipular y volver a repetir los historiales de transformaciones.

```
> Define naturals -> 0:  map (1+) naturals
> Define index (x:_) 0 -> x
  and    index (_:xs) (n+1) -> index xs n
> Law    "indmap" f xs n -> f  ⊥ ≡ ⊥  ⊢
         f (index xs n) ≡ index (map f xs) n
> Law    "indnat" n -> index naturals n ≡ n
> Proof indnat
Proof clauses of law indnat
  1.  index naturals n ≡ n
> Cases n
Proof clauses of law indnat
  * 1.1.  index naturals ⊥ ≡ ⊥
  * 1.2.  index naturals 0 ≡ 0
  * 1.3.  index naturals n1 ≡ n1 ⊢
          index naturals (n1+1) ≡ n1+1
> Unfold in 1.1 1.2
Proof clause 1.1 indnat is TRUE
Proof clause 1.2 indnat is TRUE
```

```
.....................................
> Define naturals -> 0:  map (1+) naturals
  * 1.3. index naturals n1 ≡ n1 ⊢ index naturals (n1+1) ≡ n1+1
.....................................
> Unfold in 1.3
  * 1.3. index (0 :  map (+1) naturals) n1 ≡ n1 ⊢
         index (0 + 1 :  map (+1) (map (+1) naturals)) n1 ≡ n1 +1
> Undo
  Unfold in 1.3 indnat is undone
> Unfold index _ (n1+1) in 1.3
  * 1.3. index naturals n1 ≡ n1 ⊢
         index (map (+1) naturals) n1 ≡ n1 + 1
> Apply indmap (1+) naturals n1
  * 1.3. index naturals n1 ≡ n1 ⊢
         1 + index naturals n1 ≡ n1 + 1
> Induce
Proof clause 1.3 indnat is TRUE
Law indnat is provisionally proved
> Status
Law indnat :  indmap
```

Sistema MAG

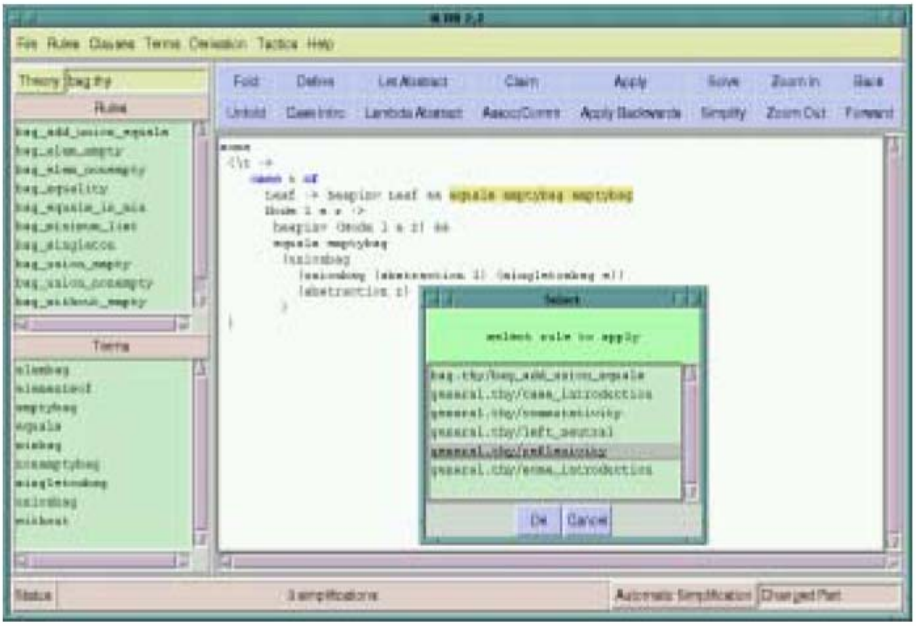
Sistema de transformación para un subconjunto de Haskell, que permite realizar encaje de patrones de segundo orden [MAG]. Para hacer el encaje utiliza backtracking y permite hacer transformaciones usando técnicas como el teorema de fusión y algunas otras. Entre sus características más notables se puede mencionar la posibilidad de escribir código activo, es decir, código que forma parte en el proceso de compilación, con instrucciones que indican al compilador las posibles optimizaciones.



Sistema ULTRA

Este sistema permite asistir a los programadores para la derivación formal de programas obtenido programas oficien desde una descripción en alto nivel o la especificación de sus operaciones [US]. El sistema usa lambda abstracción. Está escrito en el lenguaje funcional Gofra. Tiene un interfaz cómodo, una implementación fácilmente extensible, y tiene una manera fácil de expresar las reglas de transformación. A continuación podemos ver ejemplos de su funcionamiento.

```
sort :: [a] → [a]
sort xs = some (λys → issorted ys ∧
elementsof ys `equals` elementsof xs)
issorted :: [a] → Bool
issorted [ ] = True
issorted (x : xs) = x == minimum (x : xs) ∧ issorted xs

..
  _sort xs_
≡ {|unfold|} some (λys → issorted ys ∧ _elementsof ys `equals` elementsof xs_)
≡ {|apply introduce composetree]|}
some (λys → _issorted ys ∧ elementsof ys `equals` abstraction (composetree
xs)_)
≡ {|apply add neutral left |} some (λys → _True_ ∧ issorted ys ∧
elementsof ys `equals` abstraction (composetree xs))
≡ {|apply heap composetree backwards|} some (λys → heapinv _(composetree xs)_
∧ issorted ys ∧
elementsof ys `equals` abstraction _(composetree xs)_)
≡ {|λ-abstraction (with multiple selection)|}
_(λt → some (λys → heapinv t ∧ issorted ys ∧
elementsof ys `equals` abstraction t)) _(composetree xs)
≡ {|define decomposetree (that is automatically folded)|}
_decomposetree (composetree xs)_
≡ {|define heapsort (again, automatically folded)|}
heapsort xs
```



10. Conclusiones y Trabajo Futuro

Las conclusiones que se ha obtenido son las siguientes:

- El tema del que trata el proyecto es complejo, debido a esto no se ha podido realizar en su totalidad, se ha dejado varias cosas que se tenían planeados en un principio.

- Las transformaciones de lenguajes funcionales es un campo complejo, poca gente se ha dedicado a realizar sistemas que realicen transformaciones y las personas que lo han hecho lo tienen incompleto o sin terminar.

- Este proyecto, con ciertas ampliaciones, se lo podría utilizar como un trabajo de futuro, ya que, por ejemplo, los profesores podrían dar clases de programación funcional utilizando una herramienta que les fuera de mucha ayuda para explicar las diferentes fases de transformación por la que puede pasar un programa para convertirse en un programa mucho más eficiente.

11. Apéndices

11.1. Apéndice A. Gramática

En este apéndice se muestra la gramática del lenguaje fuente que se ha utilizado para realizar el parser de la aplicación. A continuación mostramos la gramática completa que se había mostrado en simplificada en el apartado 3.4. Esta grmática está acondicionada, es decir, se a eliminado la recurividad a izquierdas y se han factorizado las producciones necesarias. Esto se ha llevado a cabo para poder implementar el parser de manera adecuada con JAVACC, ya que está herramienta es un generador de analizadores predictivo recursivo descendentes.

```
options
{
      LOOKAHEAD=1;
}

PARSER_BEGIN(GramaticaML)

import java.io.*;

public class GramaticaML
{
   private static Nodo raiz;
   private static GramaticaML miParser;
   private static int cont;
   private boolean err;
   private Exception exc;
   private static int numDec;

   public GramaticaML() {}

   public boolean err(){return err;}

   public Exception exc(){return exc;}

   public Nodo raiz(){return
raiz.dameHijo(0);}

   public int numDec(){return numDec;}

   public void setCodigoFuente(String
archivoML)
   {
            ByteArrayInputStream is = new
ByteArrayInputStream
(archivoML.getBytes());
            cont=0; err=false; numDec=0;
            if (miParser == null)
            {
                miParser=new
GramaticaML(is);
            }
            else
            {
                miParser.ReInit(is);
            }
   }
//Para parsear una o varias funciones
(Decs) --> ARBOL
   public void parse (String archivoML)
throws ParseException
   {
         try
{
                  ByteArrayInputStream is =
new ByteArrayInputStream
(archivoML.getBytes());
                  raiz=new Nodo("rootNode");
                  cont=0; err=false;
numDec=0;

                  if (miParser == null)
                  {
                      miParser=new
GramaticaML(is);
                  }
                  else
                  {
                      miParser.ReInit(is);
                  }
                  //Llamamos a la primera
producción
                  miParser.listaDec(raiz);
            }
            catch(Exception e)
            {
               err=true;
               exc=e;
//System.err.println(e.getMessage());
            }
   }

}
PARSER_END(GramaticaML)
//-------------------------------------
 SKIP:
{
      "\t"
      |"\n" //{cont++;}
      |"\r"
      |" "
}


TOKEN :{ <NUMREAL: (<NUMINT> <PUNTO>
(<DIGITO>)* (<EXP>)?)|(<PUNTO> (<DIGITO>)+
(<EXP>)?)|           ((<DIGITO>)+ <EXP> )
> }
TOKEN :{ <#EXP:  <LETRAE> ("-")?
(<DIGITO>)+ >}
TOKEN :{ <NUMINT: (<DIGITO1> (<DIGITO>)*)
| <CERO> > }
TOKEN :{ <#DIGITO: <CERO> | <DIGITO1> >}
TOKEN :{ <#DIGITO1: ["1"-"9"]>}
TOKEN :{ <#CERO: "0"> }
```

```
TOKEN :{ <BOOL:  <FALSE> | <TRUE> >}
TOKEN:{<#letra:["a"-"z"]>}
TOKEN:{<#letraM:["A"-"Z"]>}

/*Operadores*/

TOKEN:{<OPA: <MAS> | <MENOS> | "||">}
TOKEN:{<OPM: "*"| "/"| "&&">}
TOKEN:{<OPR: "=="|"!="|"<"|"<="|">"|">=">}
TOKEN:
{     <GUIONBAJO: "_">
|   <BARRAVER: "|">
|     <PARAB: "(">
|     <PARC: ")">
|     <CORAB: "[">
|     <CORC: "]">
|     <ALMO: "#">
|     <PUNTO: ".">
|     <DOSPUNTOS: ":">
|     <PUNTOCOMA: ";">
|     <IGUAL: "=">
|     <COMA: ",">
|     <FLECHA: "=>">
|     <MAS: "+">
|     <#MENOS: "-">
|     <#LETRAE: "e">
|     <#NOT: "!">
|     <CONC: "++">
|     <OPINPUNTO: "·">
}
TOKEN[IGNORE_CASE]:
{     <SI: "if">
|     <SINO: "else">
|     <ENTONCES: "then">
|     <CASE: "case">
|     <OF: "of">
|     <FUN: "fun">
|     <LET: "let">
|     <IN: "in">
|     <WHERE: "where">
|     <END: "end">
|     <AND: "and">
|     <INF: "infix">
|     <INFR: "infixr">
|     <NINF: "nonfix">
|     <SIG: "sig">
|     <SIGNATURE: "signature">
|     <TYPE: "type">
|     <LOCAL: "local">
|     <INCLUDE: "include">
|     <OPEN: "open">
|     <#FALSE: "false">
|     <#TRUE: "true">
|    <AS: "as">
}

/* Palabras Reservadas y literales*/
TOKEN[IGNORE_CASE]:{<TYVAR: "int" | "bool"
| "char" | "real">}

//Identificadores

TOKEN :{<ID: (<letra>|<letraM>)
(<letra>|<letraM>|<DIGITO>)* >}

//-------------------------------------
listaDec()::=dec() <PUNTOCOMA> listaDec()
                  dec() <PUNTOCOMA>

dec::=<FUN> fun()

fun()::=fvalbind()

fvalbind()::=<ID> linea()

linea()::=latpat() <IGUAL> exp()

latpat()::=atpat() latpat()
           atpat()

atpat()::=<GUIONBAJO>
          scon()
          <ID>
          fpar()
              corpat()

corpat()::=<CORAB> <CORC>
           <CORAB> lpat()
           <CORC>

fpar()::=<PARAB> pat() <DOSPUNTOS> pat()
<PARC>
           fpar2()

fpar2()::=patSuma()
          <PARAB> (lpat() <PARC>|<PARC>)

lpat()::=pat() <COMA> lpat(nodoAux)
         pat ()

pat()::= <ID> atpat()
         atpat()
```

```
patSuma()::=<PARAB> idOscon() masKR()
<PARC>
           <PARAB> patSuma() masKR()
<PARC>

idOscon()::=<ID>
            scon(nodoAux)

masKR()::=masK() masKR()
          masK()

masK()::=<OPA> <NUMINT>

exp()::=expSimple() opRel()  expSimple()
        expSimple()
        <SI> exp() <ENTONCES> exp() <SINO>
exp(nodoAux)
        <CASE> exp(nodoAux) <OF>
match(nodoAux)

opRel()::=<OPR>

expSimple()::=termino() resExpSimple()
              termino()

resExpSimple()::=opAdd() termino()
resExpSimple()
                 opAdd() termino()

opAdd()::=<OPA>
          <CONC>
          <OPINPUNTO>

termino()::=factor() resTermino()
            factor()

resTermino()::=opMul() factor()
resTermino()
               opMul() factor()

opMul()::=<OPM>

factor()::=infexp() factor()
           infexp()

listaId()::=<ID> <COMA> listaId()
            <ID>

infexp()::=scon()
           <WHERE> pat() <IGUAL> exp()
           <ID>
           fparfact()
           corpatfact()
```

```
fparfact()::=<PARAB> exp()
                 <DOSPUNTOS> exp()
                 <PARC>
               fpar2fact()

fpar2fact()::=<PARAB> (lexp()
<PARC>|<PARC>)

lexp()::=exp() <COMA> lexp()
         exp ()

corpatfact()::=<CORAB> <CORC>
               <CORAB> lexp() <CORC>

scon()::=<NUMINT>
         <BOOL>
         <NUMREAL>

match()::=mrule() <BARRAVER> match()
          mrule()

mrule()::=pat() <FLECHA> exp()
```

11.2. Apéndice B. Ejemplos del Lenguaje de Transformación

11.2.1. Definición

Comentario Inicial	**DEFINICIÓN DE UNA FUNCIÓN (NO HACE FALTA; AL FINAL DE LA DEFINICIÓN)** `fun suma x y = x + y;`	**Nº 1**
Transformación	`def fun resta x y = x - y`	
Final	`fun suma x y = x + y;` `fun resta x y = x - y;`	
Comentario Inicial	**DEFINICIÓN DE UNA FUNCIÓN: OTRA FORMA, CON; AL FINAL** `fun suma x y = x + y;`	**Nº 2**
Transformación	`def fun resta x y = x – y;`	
Final	`fun suma x y = x + y;` `fun resta x y = x - y;`	
Comentario Inicial	**DEFINICIÓN DE UNA FUNCIÓN: EN LUGAR DE def SE PUEDE PONER define** `fun suma x y = x + y;`	**Nº 3**
Transformación	`define fun resta x y = x – y`	
Final	`fun suma x y = x + y;` `fun resta x y = x - y;`	

11.2.2. Instanciación

Comentario	**INSTANCIACIÓN DE LA VARIABLE X CON LOS PATRONES DE ENTEROS 0 Y (n+1)**	
Inicial	`fun suma x y = x + y;`	**Nº1**
Transformación	`ins 0 int x`	
Final	`fun suma 0 y = 0 + y;` `fun suma (n+1) y = (n+1) + y;`	
Comentario	**INSTANCIACIÓN DE LA VARIABLE X CON LOS PATRONES DE LISTAS [] Y (o:os)**	
Inicial	`fun suma x y = x + y;`	**Nº2**
Transformación	`ins 0 list x`	
Final	`fun suma [] y = [] + y;` `fun suma (o:os) y = (o:os) + y;`	
Comentario	**EL PATRÓN DE ENTEROS (n+1) ES LIBRE, ASÍ SI HAY UNA VARIABLE n SE SUSTITUYE POR OTRO PATRÓN LIBRE**	
Inicial	`fun suma x y n = x + y + n;`	**Nº3**
Transformación	`ins 0 int y`	
Final	`fun suma x 0 n = x + 0 + n ;` `fun suma x ( o + 1 ) n = x + ( o + 1 ) + n ;`	
Comentario	**INSTANCIACIÓN EN SECUENCIA**	
Inicial	`fun suma ( (o+1) + 1 ) y n = ( (o+1) + 1 ) + y + n ;`	**Nº4**
Transformación	`ins 0 int o`	
Final	`fun suma ( 0 + 1 ) y n = ( 0 + 1 ) + y + n ;` `fun suma ( ( p + 1 ) + 1 ) y n = ( ( p + 1 ) + 1 ) + y + n ;`	
Comentario	**INSTANCIACIÓN EN UN WHERE**	
Inicial	`fun suma x y = x + y + n` `  where n = resta x y ;`	**Nº5**
Transformación	`ins 0 int x`	

Final	`fun suma 0 y = 0 + y + n`
	`  where n = resta 0 y ;`
	`fun suma ( o + 1 ) y = ( o + 1 ) + y + n`
	`  where n = resta ( o + 1 ) y ;`	

Comentario	**INSTANCIACIÓN EN UN IF…THEN…ELSE Y USO DE REALES**	
Inicial	`fun suma x y = if (x>1) then x + y else 0.5;`	**Nº6**
Transformación	`ins 0 int x`	
Final	`fun suma 0 y = if ( 0 > 1 ) then 0 + y else 0.5 ;` `fun suma (n + 1) y = if (( n + 1 ) > 1) then ( n + 1 ) + y else 0.5 ;`	

Comentario	**INSTANCIACIÓN EN UN CASE**	

Inicial	`fun long xs = case [ ] of`
	`  [ ] => 0`
	`  | ( x : xs ) => 1 + long xs`
	`  | [ x1 , x2 ] => 1 + long [ x2 ] ;`
		Nº7

Transformación	`ins 0 list xs`	

Final	`fun long [ ] = case [ ] of`
	`  [ ] => 0`
	`  | ( x : [ ] ) => 1 + long [ ]`
	`  | [ x1 , x2 ] => 1 + long [ x2 ] ;`
	
	`fun long ( o : os ) = case [ ] of`
	`  [ ] => 0`
	`  | ( x : ( o : os ) ) => 1 + long ( o : os )`
	`  | [ x1 , x2 ] => 1 + long [ x2 ] ;`	

Comentario	**SUSTITUCIÓN DE LA VARIABLE X POR UN NÚMERO**	
Inicial	`fun suma x y = x + y;`	**Nº1**
Transformación	`sust 0 x 5`	
Final	`fun suma 5 y = 5 + y;`	

Comentario	**SUSTITUCIÓN DE LA VARIABLE X POR UNA EXPRESIÓN**	
Inicial	`fun suma x y = x + y;`	**Nº2**
Transformación	`sust 0 x (y+5)`	
Final	`fun suma (y+5) y = (y+5) + y;`	

11.2.3. Pliegue

Comentario Inicial	**ENCAJE DE PATRONES: TUPLA CON PARÁMETROS DE FUNCIÓN(a->x, b->y, c->z)**
	`fun x sd = sum ( x , y , z ) ;`
	`fun sumalog a b c = sum ( a , b , c ) ;`
		Nº 1

Transformación	`fold 0 "sum(x,y,z)" sumalog 1`	
Final	`fun x sd = sumalog x y z ;` `fun sumalog a b c = sum ( a , b , c ) ;`	

Comentario Inicial	**ENCAJE DE PATRONES: TUPLA CON LISTAS UNITARIAS (a->x, b->y, c->z)**
	`fun x sd = sum ( x , y , z ) ;`
	`fun sumalog [a] [b] [c] = sum ( a , b , c ) ;`
		Nº 2

Transformación	`fold 0 "sum(x,y,z)" sumalog 1`	
Final	`fun x sd = sumalog [ x ] [ y ] [ z ] ;` `fun sumalog [ a ] [ b ] [ c ] = sum ( a , b , c ) ;`	

Comentario Inicial	**INTENTO DE ENCAJE DE DOS NÚMEROS DIFERENTES CON LA MISMA VARIABLE**
	`fun x sd = sum ( 5 , 8 , z ) ;`
	`fun sumalog [ x ] [ x ] [ c ] = sum ( x , x , c ) ;`
		Nº 3

Transformación	`fold 0 "sum(5, 8, z)" sumalog 1`	
Final	`No encaja y no hace nada porque x no puede ser 5 y 8 a la vez`	

Comentario Inicial	**ENCAJE DEL PATRÓN LISTA Y CONSERVACIÓN DE VAR LIBRE (a->x, b->xs)**
	`fun suma x y = sum(x:xs);`
	`fun sumalog a b c= sum(a:b);`
		Nº 4

Transformación	`fold 0 "sum(x:xs)" sumalog 1`	
Final	`fun suma x y = sumalog x xs c ;` `fun sumalog a b c = sum ( a : b ) ;`	

Comentario Inicial	**ENCAJE DE LISTAS (a->x, b->xs, c->y)**
	`fun suma x xs y = sum(x:xs)+y;`
	`fun sumalog a b c= sum(a:b)+c;`
		Nº 5

Transformación	`fold 0 "sum(x:xs) + y" sumalog 1`	
Final	`fun suma x xs y = sumalog x xs y ;` `fun sumalog a b c = sum ( a : b ) + c ;`	

Comentario Inicial	**ENCAJE DE EXPRESIONES LARGAS (a->x)**
	`fun suma x y = sum x (5*2-2/9+6);`
	`fun sumalog a 5= sum a (5*2-2/9+6);`
		Nº 6

Transformación	`fold 0 "sum x (5*2-2/9+6)" sumalog 1`	
Final	`fun suma x y = sumalog x 5 ;` `fun sumalog a 5 = sum a ( 5 * 2 - 2 / 9 + 6 ) ;`	

Comentario Inicial	**PLIEGUE DENTRO DE UN WHERE (a->x)**
	`fun suma x y = sum x y`
	`  where y = sum x;`
	`fun sumalog a = sum a ;`
		Nº 7

Transformación	`fold 0 "sum x" sumalog 1`	

Final	`fun suma x y = sum x y`
	`  where y = sumalog x ;`
	`fun sumalog a = sum a ;`	

Comentario Inicial	**EJEMPLO DE NO ENCAJE DE BOOLEANOS (true!=false)**
	`fun suma x y = sum x false;`
	`fun sumalog a = sum a true;`
		Nº 8

Transformación	`fold 0 "sum x false" sumalog 1`	
Final	`No hace nada pq no encaja true con false;`	

Comentario Inicial	**EJEMPLO DE ENCAJE DE BOOLEANOS**
	`fun suma x y = sum x false;`
	`fun sumalog a = sum a false;`
		Nº 9

Transformación	`fold 0 "sum x false" sumalog 1`	
Final	`fun suma x y = sumalog x ;` `fun sumalog a = sum a false ;`	

Comentario Inicial	**EJEMPLO DE NO ENCAJE DE DOS NÚMEROS DIFERENTES CON LA MISMA VARIABLE**
	`fun suma x y= sum(x,x,z);`
	`fun sumlog a b c = sum(5,6,z);`
		Nº 10

Transformación	`fold 0 "sum (x,x,z)" sumlog 1`	
Final	`No encaja porque ya hay una clave (5->x) que no es una variable que tiene x como valor, no permite la generación de otra clave (6->x).`	

Comentario Inicial	**ENCAJE DE LISTAS CON LISTAS, Y VARIABLES CON VARIABLES**
	`fun suma x y = sum(x:xs)+y;`
	`fun sumalog a b c= sum(a:b)+c;`
		Nº 11

Transformación	`fold 0 "sum(x:xs) + y" sumalog 1`	
Final	`fun suma x y = sumalog x xs y ;` `fun sumalog a b c = sum ( a : b ) + c ;`	

Comentario Inicial	**ENCAJE DE LISTAS CON VARIABLES Y CON LISTAS FORMADAS POR VARIABLES**
	`fun suma x y= sum (x:xs) (z:zs);`
	`fun sumlog y ys k = sum (y:ys) k;`
		Nº 12

Transformación	`fold 0 "sum (x:xs) (z:zs)" sumlog 1`	
Final	`fun suma x y = sumlog x xs ( z : zs ) ;` `fun sumlog y ys k = sum ( y : ys ) k ;`	

Comentario Inicial	**EJEMPLO DE NO ENCAJE DE LISTA VACIA Y LISTA LLENA**
	`fun suma x y= sum (x:xs) (z:zs);`
	`fun sumlog y ys k = sum (y:ys) [];`
		Nº 13

Transformación	`fold 0 "sum (x:xs) (z:zs)" sumlog 1`	
Final	`No encaja porque la lista (z:zs) solo encaja con otra lista o una variable.`	

Comentario	**EVALUACIÓN DEL VALOR DE UNA VARIABLE: (a->x), (5->x) => (a->5) CUANDO APARECE (5->x), SE ASIGNA A TODAS LAS VARS QUE VALÍAN X, 5.**	
Inicial	`fun suma x= sum x x;` `fun sumlog a = sum a 5;`	**Nº 14**
Transformación	`fold 0 "sum x x" sumlog 1`	
Final	`fun suma x = sumlog 5 ;` `fun sumlog a = sum a 5 ;`	
Comentario	**EVALUACIÓN DEL VALOR DE UNA VARIABLE: (5->x), (b->x) => (b->5) CUANDO APARECE (b->x), SE REVISA SI YA LA x TIENE UN VALOR (AQUÍ 5), Y SE ACTUALIZA LA b A ESE VALOR 5.**	
Inicial	`fun suma x= sum x x;` `fun sumlog b = sum 5 b;`	**Nº 15**
Transformación	`fold 0 "sum x x" sumlog 1`	
Final	`fun suma x = sumlog 5 ;` `fun sumlog b = sum 5 b ;`	
Comentario	**EVALUACIÓN DEL VARIOS VALORES: (5->x),(b->x),(c->x) => (b->5), (c->5)**	
Inicial	`fun suma x= sum x x x;` `fun sumlog b c = sum 5 b c;`	**Nº 16**
Transformación	`fold 0 "sum x x x" sumlog 1`	
Final	`fun suma x = sumlog 5 5 ;` `fun sumlog b c = sum 5 b c ;`	
Comentario	**EJEMPLO DE GENERACIÓN DE VARIABLES LIBRES (c SE CONVIERTE EN n)**	
Inicial	`fun suma x= sum x y + c;` `fun sumlog a b c = sum a b;`	**Nº 17**
Transformación	`fold 0 "sum x y" sumlog 1`	
Final	`fun suma x = sumlog x y n + c ;` `fun sumlog a b c = sum a b ;`	
Comentario	**EJEMPLO DE VARIAS VARIABLES LIBRES (TODOS LOS PARÁMETROS DE sumlog SON LIBRES)**	
Inicial	`fun suma x = sum x y + w + z ;` `fun sumlog w z x = sum a b ;`	**Nº 18**
Transformación	`fold 0 "sum x y" sumlog 1`	
Final	`fun suma x = sumlog n o p + w + z ;` `fun sumlog w z x = sum a b ;`	
Comentario	**EJEMPLO DE TRANSFORMACIÓN DEL _ EN UNA VARIABLE LIBRE**	
Inicial	`fun suma x = sum x y + w + z ;` `fun sumlog _ z x = sum a b ;`	**Nº 19**
Transformación	`fold 0 "sum x y" sumlog 1`	
Final	`fun suma x = sumlog n o p + w + z ;` `fun sumlog _ z x = sum a b ;`	

Comentario	**PLIEGUE EN LA SEGUNDA APARICIÓN DE sum x x x**	
Inicial	`fun suma x= sum x x x + sum x x x - sum x x x;` `fun sumlog b c = sum 5 b c;`	**Nº 20**
Transformación	`fold 0 "sum x x x" 2 sumlog 1`	
Final	`fun suma x = sum x x x + sumlog 5 5 - sum x x x ;` `fun sumlog b c = sum 5 b c ;`	
Comentario	**PLIEGUE EN UNA APARICIÓN QUE NO EXISTE -> NO HACE NADA**	
Inicial	`fun suma x= sum x x x + sum x x x - sum x x x;` `fun sumlog b c = sum 5 b c;`	**Nº 21**
Transformación	`fold 0 "sum x x x" 4 sumlog 1`	
Final	`No hace nada porque no hay 4 apariciones de sum x x x.`	
Comentario	**PLIEGUE EN TODAS LAS APARICIONES POSIBLES SI NO SE ESPECIFICA EL NÚMERO DE APARICIÓN**	
Inicial	`fun suma x= sum x x x + sum x x x - sum x x x;` `fun sumlog b c = sum 5 b c;`	**Nº 22**
Transformación	`fold 0 "sum x x x" sumlog 1`	
Final	`fun suma x = sumlog 5 5 + sumlog 5 5 - sumlog 5 5 ;` `fun sumlog b c = sum 5 b c ;`	
Comentario	**PLIEGUE EN UN CASE**	

Inicial	`fun long xs = case [] of`
	`[]=>0`
	`|(x:xs)=>1+long xs`
	`|[x1,x2]=> 1+ long[x2];`
	`fun longConFold a = long a;`
		Nº 23

Transformación	`fold 0 "long xs" longConFold 1`	

Final	`fun long xs = case [ ] of`
	`[ ] => 0`
	`| ( x : xs ) => 1 + longConFold xs`
	`| [ x1 , x2 ] => 1 + long [ x2 ] ;`
	`fun longConFold a = long a ;`	

Comentario	**ENCAJE DE LISTAS EN UN PLIEGUE**	
Inicial	`fun media a b c d = suma [a,b,c,d];` `fun sumlong x xs = suma (x:xs);`	**Nº 24**
Transformación	`fold 0 "suma [a,b,c,d]" sumlong 1`	
Final	`fun media a b c d = sumlong a [ b , c , d ] ;` `fun sumlong x xs = suma ( x : xs ) ;`	

11.2.4. Despliegue

Comentario	**DESPLIEGUE DE UNA FUNCIÓN SIMIPLE**	
Inicial	`fun media l = sum l / long l ;` `fun sum [] = 0.0;`	**Nº 1**
Transformación	`unf 0 sum 1`	
Final	`fun media l = 0.0 / long l ;` `fun sum [ ] = 0.0 ;`	
Comentario	**DESPLIEGUE EN LA SEGUNDA APARICIÓN DE sum l**	
Inicial	`fun media l = sum l / long l + sum l * long l;` `fun sum [] = 0.0;`	**Nº 2**
Transformación	`unf 0 2 sum 1`	
Final	`fun media l = sum l / long l + 0.0 * long l;` `fun sum [ ] = 0.0 ;`	
Comentario	**DESPLIEGUE EN TODAS LAS APARICIONES DE sum l**	
Inicial	`fun media l = sum l / long l + sum l * long l;` `fun sum [] = 0.0;`	**Nº 3**
Transformación	`unf 0 sum 1`	
Final	`fun media l = 0.0 / long l + 0.0 * long l;` `fun sum [ ] = 0.0 ;`	
Comentario	**GENERA NUEVAS VARIABLES LIBRES, YA QUE EN (m+5) AL PRINCIPIO, m ES LIBRE. SE GENERAN VARIABLES LIBRES DIFERENTES n Y o.**	
Inicial	`fun media m = sum m / long m + sum m * long m;` `fun sum [] = (m+5);`	**Nº 4**
Transformación	`unf 0 sum 1`	
Final	`fun media m = ( ( n + 5 ) ) / long m + ( ( o + 5 ) ) * long m ;` `fun sum [ ] = ( m + 5 ) ;`	
Comentario	**LA VARIABLE LIBRE u SIGUE SIENDO u PORQUE NO ESTÁ LIGADA EN LA 1ª FUNCIÓN. LA EXPRESIÓN (m+u) SE PONE ENTRE PARÉNTESIS.**	
Inicial	`fun media m = sum m / long m + sum m * long m;` `fun sum a = a + u;`	**Nº 5**
Transformación	`unf 0 sum 1`	
Final	`fun media m = ( m + u ) / long m + ( m + u ) * long m ;` `fun sum a = a + u ;`	
Comentario	**DESPLIEGUE EN UNA LISTA UNITARIA**	
Inicial	`fun media m = [sum m] / long m + sum m * long m;` `fun sum a = a + u;`	**Nº 6**
Transformación	`unf 0 sum 1`	
Final	`fun media m = [ ( m + u ) ] / long m + ( m + u ) * long m ;` `fun sum a = a + u ;`	

Comentario Inicial — **DESPLIEGUE EN UN IF…THEN…ELSE** — **Nº 7**

```
fun media m = if true then sum m / long m else sum m * long m;
fun sum a = a + u;
```

Transformación

```
unf 0 sum 1
```

Final

```
fun media m = if true then (m + u) / long m else ( m + u ) * long m ;
fun sum a = a + u ;
```

Comentario Inicial — **ENCAJE DE BOOLEANOS Y CONSERVACIÓN DE 2 VARIABLES LIBRES** — **Nº 8**

```
fun media m = sum false / long m + sum true * long m;
fun sum true = a + u;
```

Transformación

```
unf 0 sum 1
```

Final

```
fun media m = sum false / long m + ( a + u ) * long m ;
fun sum true = a + u ;
```

Comentario Inicial — **SE APLICA EL ENCAJE A TODAS LAS FUNCIONES QUE SE PUEDE** — **Nº 9**

```
fun media m = sum ((sum m),5) / long m + sum true * long m;
fun sum a = a + u;
```

Transformación

```
unf 0 sum 1
```

Final

```
fun media m = ((((m + u)),5) + u ) / long m + ( true + u ) * long m ;
fun sum a = a + u ;
```

Comentario Inicial — **DESPLIEGUE EN UN WHERE** — **Nº 10**

```
fun media l = sum l / long l + mult
      where mult= sum l * long l;
fun sum [] = 0.0;
```

Transformación

```
unf 0 sum 1
```

Final

```
fun media l = 0.0 / long l + mult
      where mult = 0.0 * long l ;
fun sum [ ] = 0.0 ;
```

Comentario Inicial — **DESPLIEGUE EN UN CASE** — **Nº 11**

```
fun long xs = case [] of
                        []=>0
                        |(x:xs)=>1+long xs
                        |[x1,x2]=> 1+ long[x2];
fun long a = a;
```

Transformación

```
unf 0 long 1
```

Final

```
fun long xs = case [ ] of
                        [ ] => 0
                        | ( x : xs ) => 1 + xs
                        | [ x1 , x2 ] => 1 + [ x2 ] ;
fun long a = a ;
```

Comentario	**EVALUACIÓN DE UNA EXPRESIÓN EN UN DESPLIEGUE**	
Inicial	`fun media x = suma 8;` `fun suma (n+2)=n;`	**Nº 12**
Transformación	`unf 0 suma 1`	
Final	`fun media x = 6 ;` `fun suma ( n + 2 ) = n ;`	
Comentario	**ENCAJE DE LISTAS DE DIFERENTES TIPOS EN UN DESPLIEGUE**	
Inicial	`fun media a b c d = suma [a,b,c,d];` `fun suma (x:xs)=concat x +concat xs;`	**Nº 13**
Transformación	`unf 0 suma 1`	
Final	`fun media a b c d = ( concat a + concat [ b , c , d ] ) ;` `fun suma ( x : xs ) = concat x + concat xs ;`	

11.2.5. Abstracción

Comentario	**ABSTRACCIÓN DE UNA FUNCIÓN SIMPLE**	
Inicial	`fun media m = sum ( m , 5 ) / long m + sum true * long m;`	**Nº 1**
Transformación	`abs 0 "sum (m,5)"`	
Final	`fun media m = n / long m + sum true * long m` `  where n = sum ( m , 5 ) ;`	
Comentario	**ABSTRACCIÓN DE LA SEGUNDA APARICIÓN DE LA FUNCIÓN**	
Inicial	`fun media m = sum m + sum m + sum m;`	**Nº 2**
Transformación	`abs 0 "sum m" 2`	
Final	`fun media m = sum m + n + sum m` `  where n = sum m ;`	
Comentario	**ABSTRACCIÓN DE UNA APARICIÓN QUE NO EXISTE**	
Inicial	`fun media m = sum m + sum m + sum m;`	**Nº 3**
Transformación	`abs 0 "sum m" 4`	
Final	`No hace nada porque no hay 4 apariciones de sum m.`	
Comentario	**ABSTRACCIÓN DE TODAS LAS APARICIONES QUE HAY**	
Inicial	`fun media m = sum m + sum m + sum m;`	**Nº 4**
Transformación	`abs 0 "sum m"`	
Final	`fun media m = n + n + n` `  where n = sum m ;`	
Comentario	**ABSTRACCIÓN DE VARIAS COSAS A LA VEZ**	
Inicial	`fun media m = sum m + long l;`	**Nº 5**
Transformación	`abs 0 "sum m" "long l"`	
Final	`fun media m = n + o` `  where ( n , o ) = ( sum m , long l ) ;`	
Comentario	**NO SE HACE ABSTRACCIÓN DENTRO DEL WHERE (WHERE ANIDADO)**	
Inicial	`fun media m = sum m + long l + n` `  where n = sum m + 5;`	**Nº 6**
Transformación	`abs 0 "sum m"`	
Final	`fun media m = o + long l + n` `  where ( n , o ) = ( sum m + 5 , sum m ) ;`	
Comentario	**SE PUEDE ABSTRAER VARIABLES AUXILIARES DENTRO DEL WHERE**	
Inicial	`fun media m = sum m + long l + n` `  where n = sum m + 5;`	**Nº 7**
Transformación	`abs 0 "n"`	
Final	`fun media m = sum m + long l + o` `  where ( n , o ) = ( sum m + 5 , n ) ;`	

Comentario **SE GENERA UNA VARIABLE LIBRE o YA QUE LA n ESTÁ LIGADA**

Inicial
```
fun media m n = sum m + long n;
```
Nº 8

Transformación
```
abs 0 "sum m"
```

Final
```
fun media m n = o + long n
      where o = sum m ;
```

Comentario **NO SE ABSTRAE sum m (m : n) YA QUE sum TIENE 2 ARGUMENTOS Y NO 1**

Inicial
```
fun media m = ((sum m,m),89) + sum m (m:sum m);
```
Nº 9

Transformación
```
abs 0 "sum m"
```

Final
```
fun media m = ( ( n , m ) , 89 ) + sum m ( m : n )
      where n = sum m ;
```

Comentario **ABSTRACCIÓN EN UN IF…THEN…ELSE**

Inicial
```
fun media m = if(sum m) then sum m + 1 else 7;
```
Nº 10

Transformación
```
abs 0 "sum m"
```

Final
```
fun media m = if ( n ) then n + 1 else 7
      where n = sum m ;
```

Comentario **ABSTRACCIÓN COMPLEJA**

Inicial
```
fun media m = [sum m] + ((sum m,m),89) + sum (m:sum m);
```
Nº 11

Transformación
```
abs 0 "sum m"
```

Final
```
fun media m = [ n ] + ( ( n , m ) , 89 ) + sum ( m : n )
      where n = sum m ;
```

Comentario **ABSTRACCION DEL CASE**

Inicial
```
fun long xs = case [] of
       []=>0
      |(x:xs)=>1+long xs
      |[x1,x2]=> 1+ long[x2];
```
Nº 12

Transformación
```
 abs 0 "long xs"
```

Final
```
fun long xs = case [ ] of
     [ ] => 0
     | ( x : xs ) => 1 + n
     | [ x1 , x2 ] => 1 + long [ x2 ]
where n = long xs ;
```

Comentario **ABSTRACCION DEL CASE PARA VARIAS ABSTRACCIONES**

Inicial **Nº 13**

```
fun nomf xs = case 4 of
          1=> x+w
         |2=> long xs
         |3=> suma xs
         |4=> numAp xs;
```

Transformación

```
 abs 0 "long xs" "suma xs" "numAp xs"
```

Final

```
fun nomf xs = case 4 of
          1 => x + w
       | 2 => n
       | 3 => o
       | 4 => p
where ( n , o , p ) = ( long xs , suma xs , numAp xs ) ;
```

Comentario **ABSTRACCION DEL CASE PARA VARIAS ABSTRACCIONES**

Inicial **Nº 14**

```
fun nomf xs = case 4 of
          1=> 5+3
         |2=> 4
         |3=> 5+x
         |4=> numAp xs;
```

Transformación

```
abs 0 "5+3" "4"
```

Final

```
fun nomf xs = case 4 of
          1 => n
       | 2 => o
       | 3 => 5+x
       | 4 => numAp xs
where ( n , o ) = ( 5 + 3 , 4 ) ;
```

11.2.6. Leyes

Definir ley

Comentario	**DEFINICIÓN DE UNA LEY**
Transformación	`defley conmutatividad a+b <=> b+a`
Final	En la ventana de leyes aparece la nueva ley conmutatividad.

Usar ley

Comentario	**USAR UNA LEY PREDEFINIDA**	
Inicial	`fun media m = sum (if (x>0) then (m+1) else (m-1));`	**Nº 1**
Transformación	`ul leyif 0`	
Final	`fun media m = if ( x > 0 ) then sum (m+1) else sum (m-1) ;`	
Comentario	**USAR LA LEY DE ASOCIATIVIDAD**	
Inicial	`fun suma x y z = (x+ y) + z;`	**Nº 2**
Transformación	`ul asoc+ 0 1 di "(x+y)+z"`	
Final	`fun suma x y z = x + ( y + z ) ;`	
Comentario	**USAR LA LEY DE CONMUTATIVIDAS**	
Inicial	`fun suma x y  = x + y;`	**Nº 3**
Transformación	`ul conmut+ 0 1 id "x+y"`	
Final	`fun suma x y = y + x ;`	
Comentario	**USAR UNA LEY DEL ELMENTO NEUTRO DE LA SUMA**	
Inicial	`fun suma x = x + 0;`	**Nº 4**
Transformación	`ul neutro+ 0 1 id "x+0"`	
Final	`fun suma x = x ;`	
Comentario	**USAR UNA LEY DEL ELMENTO NEUTRO DE LA MULTIPLICACIÓN**	
Inicial	`fun funcion x = x *1;`	**Nº 5**
Transformación	`ul neutro* 0 1 id "x*1"`	
Final	`fun funcion x = x;`	

Constant folding

Comentario	**CONSTANT FOLDING EN LA LINEA 0**	
Inicial	fun suma1 x = 1+2; fun suma2 y = 3*4;	**Nº 1**
Transformación	cf 0	
Final	fun suma1 x = 3; fun suma2 y = 3*4;	
Comentario	**CONSTANT FOLDING EN LA LINEA 3**	
Inicial	fun suma1 x = 1+2; fun suma2 y = 3*4;	**Nº 2**
Transformación	cf 3	
Final	No hace nada porque no hay línea 3.	
Comentario	**CF DE UN IF: SE EVALÚA LA CONDICIÓN, SI ES TRUE, CALCULA LA PARTE THEN**	
Inicial	fun suma1 x = if(true||x)then 4/2*6 else 9;	**Nº 3**
Transformación	cf 0	
Final	fun suma1 x = 12 ;	
Comentario	**CF DE UN IF: SE EVALÚA LA CONDICIÓN, SI ES FALSE, CALCULA LA PARTE ELSE**	
Inicial	fun suma1 x = if(false && x)then 4/2*6 else 9;	**Nº 4**
Transformación	cf 0	
Final	fun suma1 x = 9;	
Comentario	**CF DE UN IF: SI NO SE PUEDE EVALUAR LA CONDICIÓN, NO HACE NADA**	
Inicial	fun suma1 x = if (x)then 4/2*6 else 9;	**Nº 5**
Transformación	cf 0	
Final	No hace nada porque no puede evaluar la condición del if.	
Comentario	**CF DE UN IF: SIMPLIFICA LO QUE PUEDE UNA CONDICIÓN**	
Inicial	fun suma1 x = if (true && x) then 7 else 8;	**Nº 6**
Transformación	cf 0	
Final	fun suma1 x = if (x) then 7 else 8;	
Comentario	**CF REALIZA CÁLCULOS TENIENDO EN CUENTA LAS PRIORIDADES DE OPERADORES**	
Inicial	fun suma1 x = 5+4+24/6*2;	**Nº 7**
Transformación	cf 0	
Final	fun suma1 x = 17;	

Comentario	**CF REALIZA CÁLCULOS CON OPERADORES RELACIONALES**	
Inicial	`fun suma1 x = if(true&&(4==5))then 2 else 89;`	**Nº 8**
Transformación	`cf 0`	
Final	`fun suma1 x = 89;`	
Comentario	**CF NO SE REALIZA SI HAY VARIABLES EN LA EXPRESIÓN**	
Inicial	`fun suma1 x = 3+4/x*2;`	**Nº 9**
Transformación	`cf 0`	
Final	`No hace nada porque hay una variable en la expresión.`	
Comentario	**CF SÍ HACE LA REDUCCIÓN DE EXPRESIONES BOOLEANAS**	
Inicial	`fun suma1 x = true && x;`	**Nº 10**
Transformación	`cf 0`	
Final	`fun suma1 x = (x);`	
Comentario	**CF SÍ HACE LA REDUCCIÓN DE EXPRESIONES BOOLEANAS**	
Inicial	`fun suma1 x = true || x;`	**Nº 11**
Transformación	`cf 0`	
Final	`fun suma1 x = true;`	
Comentario	**OTRO EJEMPLO DE REDUCCIÓN DE EXPRESIONES BOOLEANAS**	
Inicial	`fun suma1 x = false && c;`	**Nº 12**
Transformación	`cf 0`	
Final	`fun suma1 x = false;`	
Comentario	**REDUCCIÓN DE LISTA NO VACIA**	
Inicial	`fun suma x = (4+2:(5+3:xs));`	**Nº 13**
Transformación	`cf 0`	
Final	`fun suma x = ( 6 : ( 8 : xs ) ) ;`	
Comentario	**REDUCCIÓN DE TUPLAS NO VACIAS**	
Inicial	`fun suma x = (5+6,7,x,3+2);`	**Nº 14**
Transformación	`cf 0`	
Final	`fun suma x = ( 11 , 7 , x , 5 9 ;`	
Comentario	**REDUCCIÓN DE EXPRESIONES DENTRO DEL WHERE**	
Inicial	`fun suma x = (n,7+1,x,t)where (n,t)=(6+3,5+8);`	**Nº 15**
Transformación	`cf 0`	
Final	`fun suma x = ( n , 8 , x , t )` `where ( n , t ) = ( 9 , 13 ) ;`	

Comentario **REDUCCIÓN PARA EL CASE**

Inicial
```
fun s d = case 5 of
              5=>vt
             |6=>3+2
             |7=> 6;
```
Nº 16

Transformación
```
cf 0
```

Final
```
fun s d = vt;
```

Comentario **REDUCCIÓN PARA EL CASE**

Inicial
```
fun s d = case 8 of

   6=>vt
  |8=>3+2
  |3=> 6;
```
Nº 17

Transformación
```
 cf 0
```

Final
```
fun s d = 5;
```

Comentario **REDUCCIÓN PARA EL CASE**

Inicial
```
fun s d = (case (4*2) of

       6=>vt
      |8=>d+s
      |3=> 6) +5;
```
Nº 18

Transformación
```
 cf 0
```

Final
```
fun s d = (d + s) + 5;
```

Comentario **REDUCCIÓN PARA EL CASE**

Inicial
```
fun long xs = case [] of
       []=>0
      |(x:xs)=>1+long xs
      |[x1,x2]=> 1+[x2];
```
Nº 19

Transformación
```
 cf 0
```

Final
```
fun long xs = 0;
```

11.3. Apéndice C. Repertorio de Ejemplos de Transformación

A continuación se muestran dos ejemplos con todo detalle, en los que se puede apreciar el resultado de cada una de las instrucciones de transformación. En los siguientes ejemplos sólo se mostrarán el programa de partida, las instrucciones de transformación y el programa transformado. Si se quiere ver en detalle la transformación basta con hacer uso de la aplicación.

Fusión Horizontal

```
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
1. {def fun sumlong xs = (sum xs , long
xs)}
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
   5. fun sumlong xs = ( sum xs , long
      xs ) ;
2. {ins 5 list xs}
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
   5. fun sumlong [ ] = ( sum [ ],long [
      ]) ;
   6. fun
      sumlong(o:os)=(sum(o:os),long(o:os
      ) );
   (*)fun sumlong xs = ( sum xs , long
   xs ) ;
3. {unf 5 sum 1; unf 5 long 3}
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
   5. fun sumlong [ ] = ( 0, 0) ;
   6. fun
      sumlong(o:os)=(sum(o:os),long(o:os
      ) );
   (*)fun sumlong xs = ( sum xs , long
   xs ) ;
4. {unf 6 sum 2; unf 6 long 4}
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
   5. fun sumlong [ ] = ( 0, 0) ;
   6. fun sumlong ( o : os ) = ((o + sum
      os), ( 1 + long os ) ) ;
   (*)fun sumlong xs = ( sum xs , long
   xs ) ;
5. { abs 6 "sum os" "long os" }
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
   5. fun sumlong [ ] = ( 0, 0) ;
   6. fun sumlong (o:os)=((o+n) , ( 1 +
      p ) )
            where (n,p)=(sum os , long os
   ) ; (*)fun sumlong xs = ( sum xs ,
   long xs ) ;
6. {copy T2 0
   fold 6 "(sum os, long os)" sumlong 7}
   0. fun media l = sum l / long l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs
      ;
   5. fun sumlong [ ] = ( 0, 0) ;
   6. fun sumlong (o:os)=((o+n), (1+p))
       where ( n , p ) = sumlong os ;
(*)fun sumlong xs = (sum xs, long xs ) ;

7. { abs 0 "sum l" "long l"}
   0. fun media l = n / o
       where (n,o)=(sum l, long l ) ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) =1 + long xs ;
   5. fun sumlong [ ] = ( 0, 0) ;
   6. fun sumlong (o:os)=((o+n),(1+p ) )
```

```
            where ( n,p ) = sumlong os ;
  (*)fun sumlong xs = (sum xs,long xs);
8. {copy T2 0
 fold 0 "(sum l, long l)" sumlong 7}
   0. fun media l = n / o
            where (n,o)= sumlong l ;
   1. fun sum [ ] = 0 ;
   2. fun sum ( x : xs ) = x + sum xs ;
   3. fun long [ ] = 0 ;
   4. fun long ( x : xs ) = 1 + long xs;
   5. fun sumlong [ ] = ( 0, 0) ;
   6. fun sumlong (o:os)=((o+n),(1+p) )
            where ( n , p ) = sumlong os;
  (*)fun sumlong xs =(sum xs,long xs ) ;
```

Promoción de Filtros

```
0. fun foldr f z []= z;
1. fun foldr f z (x:xs)=f x (foldr f z
   xs);
2. fun map f []=[];
3. fun map f (x:xs)= (f x:map f xs);
4. fun sumsq xs = foldr mas 0 (map sq
   xs);
5. fun mas x y= x+y;
6. fun sq x = x*x;

1. {define fun sumsq1 xs = sumsq xs}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 xs = sumsq xs ;
7. fun sq x = x * x ;

2. { ins 6 list xs}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = sumsq ( o :
   os ) ;
8. fun sq x = x * x ;

3. { unf 7 sumsq 4}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = foldr mas 0 (
   map sq ( o : os ) ) ;
8. fun sq x = x * x ;

4. { unf 8 map 3}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = foldr mas 0 (
   ( sq o : map sq os ) ) ;
8. fun sq x = x * x ;

5. { ul quitaPar 7 1 "(sq o : map sq
os)"}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = foldr mas 0 (
   sq o : map sq os ) ;
8. fun sq x = x * x ;

6. { ul ponePar 7 1 "sq o";
    ul ponePar 7 1 "map sq os"}
0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
```

```
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = foldr mas 0 (
   (sq o) : (map sq os) ) ;
8. fun sq x = x * x ;

7. { unf 7 foldr 1}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = mas ( sq o )
   ( foldr mas 0 ( map sq os ) ) ;
8. fun sq x = x * x ;

8. {fold 7 "foldr mas 0 (map sq os)"
sumsq 4}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = mas ( sq o )
   ( sumsq os ) ;
8. fun sq x = x * x ;

9. { defley sumsqLey sumsq a <=> sumsq1
a;
ul sumsqLey 7 1 "sumsq os"}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = mas ( sq o )
   ( sumsq1 os ) ;
8. fun sq x = x * x ;

10. { unf 7 sq 8}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = mas ( ( o * o
   ) ) ( sumsq1 os ) ;
8. fun sq x = x * x ;

11. { ul quitaPar 7 1 "(o * o)"}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs ) ;
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = mas ( o * o )
   ( sumsq1 os ) ;
8. fun sq x = x * x ;

12. { unf 7 mas 5}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
7. fun sumsq1 ( o : os ) = ( ( o * o ) +
   ( sumsq1 os ) ) ;
8. fun sq x = x * x ;

13.{ul quitaPar 7 1
        "((o*o)+(sumsq1 os))"}

0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = sumsq [ ] ;
```

```
7. fun sumsq1 ( o : os ) = ( o * o ) + (
   sumsq1 os ) ;
8. fun sq x = x * x ;
```

```
14. { unf 6 sumsq 4}
```

```
0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = foldr mas 0 ( map sq
   [ ] ) ;
7. fun sumsq1 ( o : os ) = ( o * o ) + (
   sumsq1 os ) ;
8. fun sq x = x * x ;
```

```
15. { unf 6 map 2}
```

```
0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = foldr mas 0 ( [ ] )
   ;
```

```
7. fun sumsq1 ( o : os ) = ( o * o ) + (
   sumsq1 os ) ;
8. fun sq x = x * x ;
```

```
16. { ul quitaPar 6 1 "([])"}
```

```
0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = foldr mas 0 [ ] ;
7. fun sumsq1 ( o : os ) = ( o * o ) + (
   sumsq1 os ) ;
8. fun sq x = x * x ;
```

```
17. { unf 6 foldr 0}
```

```
0. fun foldr f z [ ] = z ;
1. fun foldr f z ( x : xs ) = f x (
   foldr f z xs ) ;
2. fun map f [ ] = [ ] ;
3. fun map f ( x : xs ) = ( f x : map f
   xs );
4. fun sumsq xs = foldr mas 0 ( map sq
   xs ) ;
5. fun mas x y = x + y ;
6. fun sumsq1 [ ] = 0 ;
7. fun sumsq1 ( o : os ) = ( o * o ) + (
   sumsq1 os ) ;
8. fun sq x = x * x ;
```

Recursión final (caso general)

```
fun f x = if triv x then base x else  a x + f (t x);

Transformación:
---------------
def fun g ac x = ac + f x
unf 1 f 0
ul leyif2 1 1 id "ac + (if triv x then base x else a x + f ( t x ) )"
ul asoc+ 1 1 DI "ac + ( a x + f ( t x ) )"
def fun g ac x = ac + f x
fold 1 "(ac + a x)+ f (t x)" g 2
def fun f x = z + f x
fold 3 "z + f x" g 2

Resultado:
----------
fun f x = if triv x then base x
          else a x + f ( t x ) ;
```

```
fun g ac x =
          if triv x then ac + ( base x )
          else g ( ac + a x ) ( t x ) ;
fun g ac x = ac + f x ;
fun f x = g z x ;
```

Factorial (caso particular de recursión final)

```
fun fact n = if (n==0)then 1 else n * fact (n-1);

Transformación:
---------------
def fun g ac n = ac * fact n
unf 1 fact 0
ul leyif3 1 1 id "ac * (if (n==0) then 1 else n * fact ( n-1 ) )"
ul asoc* 1 1 DI "ac * ( n* fact (n-1) )"
def fun g ac n = ac * fact n
fold 1 "(ac * n)* fact (n-1)" g 2
def fun fact n = 1 * fact n
fold 3 "1 * fact n" g 2
ul quitaPar 1 1 id "(1)"
ul neutro* 1 1 id "ac * 1"

Resultado:
----------
fun fact n =
          if ( n == 0 ) then 1
          else n * fact ( n - 1 ) ;
fun g ac n =
          if ( n == 0 ) then ac
          else g ( ac * n ) ( n - 1 ) ;
fun g ac n = ac * fact n ;
fun fact n = g 1 n ;
```

Deforestación

```
fun f a b = suma (inter a b);
fun inter a b = if (a>b) then [] else (a: inter (a+1) b);
fun suma [] = 0;
fun suma (x:xs) = x + suma xs;

Transformación:
---------------
unf 0 inter 1
ul quitaPar 0 1 id "( ( if ( a > b ) then [ ] else ( a : inter ( a + 1 )
b ) ) )"
ul leyif 0 1 id "suma (  if ( a > b ) then [ ] else ( a : inter ( a + 1 )
b ) )"
ul quitaPar 0 1 id "([])"
ul quitaPar 0 1 id "(( a : inter ( a + 1 ) b )) "
```

```
unf 0 suma 2
ul ponPar 0 1 id "inter (a+1) b"
unf 0 suma 3
ul quitaPar 0 1 id "(a+suma(inter (a+1) b ))"
def fun f a b = suma (inter a b)
fold 0 1 "suma (inter (a+1) b)" f 4

Resultado:
----------
fun f a b = (
          if ( a > b ) then 0
          else a + f ( a + 1 ) b ) ;
fun inter a b =
          if ( a > b ) then [ ]
          else ( a : inter ( a + 1 ) b ) ;
fun suma [ ] = 0 ;
fun suma ( x : xs ) = x + suma xs ;
fun f a b = suma ( inter a b ) ;
```

Tuplamiento(1)

```
fun fib 0 = 1;
fun fib 1 = 1;
fun fib (n+2) = fib n + fib (n+1);
fun fibPar n= (fib n, fib (n+1));

Transformación:
---------------
ins 3 int n
cf 3
ul quitaPar 3 1 id "(1)"
unf 3 fib 0
unf 3 fib 1
ul asoc+ 4 1 id "(o+1)+1"
cf 4
ul quitaPar 4 1 id "(2)"
unf 4 fib 2
ul quitaPar 4 1 id "( fib o + fib ( o + 1 ) )"
abs 4 "fib o" "fib (o+1)"
copy t1 0
fold 4 1 "(fib o,fib(o+1))" fibPar 5

Resultado:
----------
fun fib 0 = 1 ;
fun fib 1 = 1 ;
fun fib ( n + 2 ) = fib n + fib ( n + 1 ) ;
fun fibPar 0 = ( 1 , 1 ) ;
fun fibPar ( o + 1 ) = ( p , n + p )
          where ( n , p ) = fibPar o ;
```

```
fun fibPar n = ( fib n , fib ( n + 1 ) ) ;
```

Tuplamiento(2)

```
fun fact 0=1;
fun fact (n+1)= (n+1)*fact n;
fun flist 0 = [];
fun flist (n+1) = (fact(n+1):flist n);
fun g n = (fact (n+1), flist n);

Transformación:
---------------
ins 4 int n
ul neutro+I 4 1 ID "0+1"
ul quitaPar 4 1 ID "(1)"
unf 4 flist 2
unf 4 fact 1
ul neutro+I 4 1 ID "0+1"
ul quitaPar 4 1 ID "(1)"
ul neutro*I 4 1 ID "1*fact 0"
unf 4 fact 0
ul quitaPar 4 1 ID "(1)"
unf 5 flist 3
ul quitaPar 5 1 ID "((fact (o+1):flist o))"
unf 5 1 fact 1
ul asoc+ 5 1 ID "(o+1)+1"
cf 5
ul quitaPar 5 1 ID "(2)"
abs 5 "fact(o+1)" "flist o"
     def fun g n = (fact (n+1),flist n)
fold 5 "(fact(o+1),flist o)" g 6
abs 3 "fact(n+1)" "flist n"
fold 3 "(fact(n+1),flist n)" g 6

Resultado:
----------
fun fact 0 = 1 ;
fun fact ( n + 1 ) = ( n + 1 ) * fact n ;
fun flist 0 = [ ] ;
fun flist ( n + 1 ) = ( o : p )
     where ( o , p ) = g n ;
fun g 0 = ( 1 , [ ] ) ;
fun g ( o + 1 ) = ( ( ( o + 2 ) * n ) , ( n : p ) )
     where ( n , p ) = g o ;
fun g n = ( fact ( n + 1 ) , flist n ) ;
```

Especialización y evaluación parcial

```
fun exp a 0 = 1;
fun exp a (n+1) = a exp a n;
```

```
Transformación:
---------------
st 0 a 2
st 1 a 2

Resultado:
----------
fun exp 2 0 = 1 ;
fun exp 2 ( n + 1 ) = 2 exp 2 n ;
```

Optimización de recorrido

```
fun double [ ] = [ ] ;
fun double (x:xs) = (( 2 * x ) : ( double xs )) ;
fun concat [ ] l = l ;
fun concat ( x : xs ) l = ( x : ( concat xs l )) ;
fun f l1 l2 = double ( concat l1 l2 ) ;

Transformación:
---------------
ins 4 list l1
unf 4 concat 2
ul quitaPar 4 1 ID "(l2)"
unf 5 concat 3
ul quitaPar 5 1 ID "((o:(concat os l2)))"
unf 5 double 1
ul quitaPar 5 1 ID "(((2*o):(double (concat os l2))))"
copy t1 0
fold 5 1 "double (concat os l2)" f 6

Resultado:
----------
fun double [ ] = [ ] ;
fun double ( x : xs ) = ( ( 2 * x ) : ( double xs ) ) ;
fun concat [ ] l = l ;
fun concat ( x : xs ) l = ( x : ( concat xs l ) ) ;
fun f [ ] l2 = double l2 ;
fun f ( o : os ) l2 = ( ( 2 * o ) : ( f os l2 ) ) ;
fun f l1 l2 = double ( concat l1 l2 ) ;
```

Optimización de espacio

```
fun lin (tip n)=(n:[]);
fun lin (tree (s,t)) = compos (lin s, lin t);
fun fpos [] = 0;
fun fpos (0:L) = fpos L;
fun fpos ((n+1):L)=n+1;
fun g (tip 0) = fpos (lin (tip 0));
fun g (tip (n+1)) = fpos (lin (tip (n+1)));
```

```
Transformación:
---------------
unf 5 lin 0
ul quitaPar 5 1 ID "((0:[]))"
unf 5 fpos 3
unf 5 fpos 2
unf 6 lin 0
ul quitaPar 6 1 ID "(((n+1):[]))"
unf 6 fpos 4
ul quitaPar 6 1 ID "(n+1)"

Resultado:
----------
fun lin ( tip n ) = ( n : [ ] ) ;
fun lin ( tree ( s , t ) ) = compos ( lin s , lin t ) ;
fun fpos [ ] = 0 ;
fun fpos ( 0 : L ) = fpos L ;
fun fpos ( ( n + 1 ) : L ) = n + 1 ;
fun g ( tip 0 ) = 0 ;
fun g ( tip ( n + 1 ) ) = n + 1 ;
```

Transformación Recursivo-Iterativa

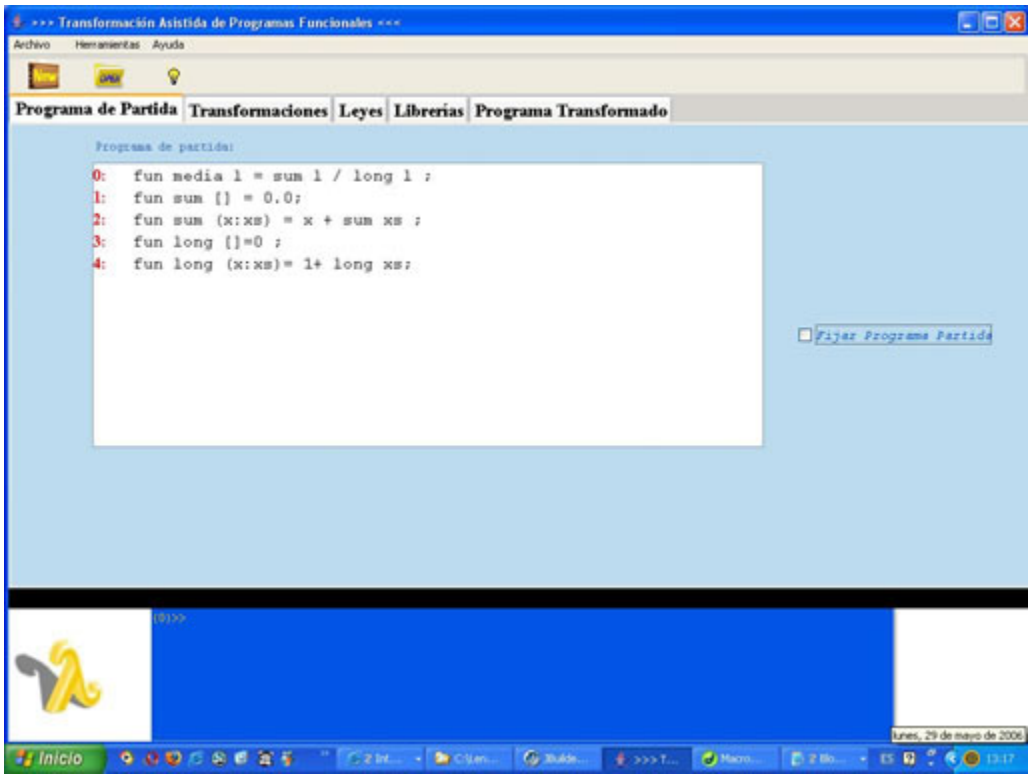
```
fun length []=0;
fun length (a:xs) = 1+length xs;
fun l n xs= n + length xs;

Transformación:
---------------
ins 2 list xs
unf 2  length 0
ul neutro+ 2 1 ID "n+0"
unf 3 length 1
ul asoc+ 3 1 DI "n+(1+length ps)"
copy t1 0
fold 3 "(n+1)+length ps" l 4

Resultado:
----------
fun length [ ] = 0 ;
fun length ( a : xs ) = 1 + length xs ;
fun l n [ ] = n ;
fun l n ( p : ps ) = l ( n + 1 ) ps ;
fun l n xs = n + length xs ;
```

11.4. Apéndice D. Manual de usuario

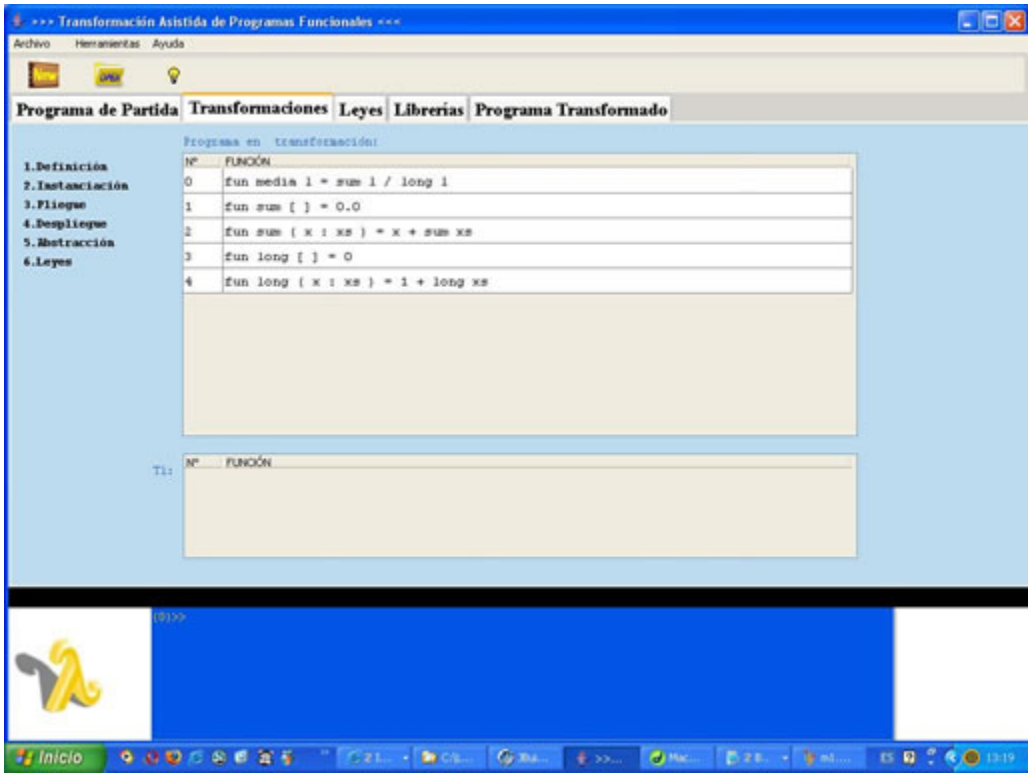
11.4.1. Descripción de las pestañas

Al abrir la aplicación aparecen 5 pestañas:

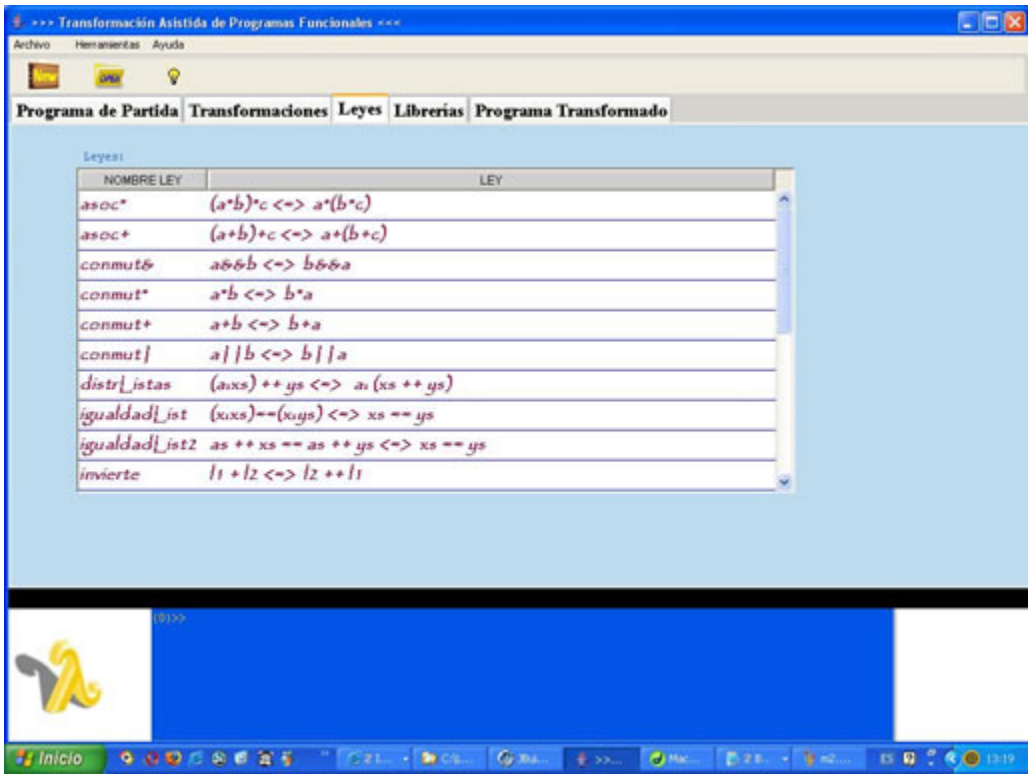
Programa de Partida, que es donde se pega el programa de partida. Ese programa no se modificará a lo largo de una transformación, y siempre se podrá consultar o empezar una transformación de nuevo pulsando en Fijar Programa Partida, que es lo que compilará el programa, generando el correspondiente árbol sintáctico.



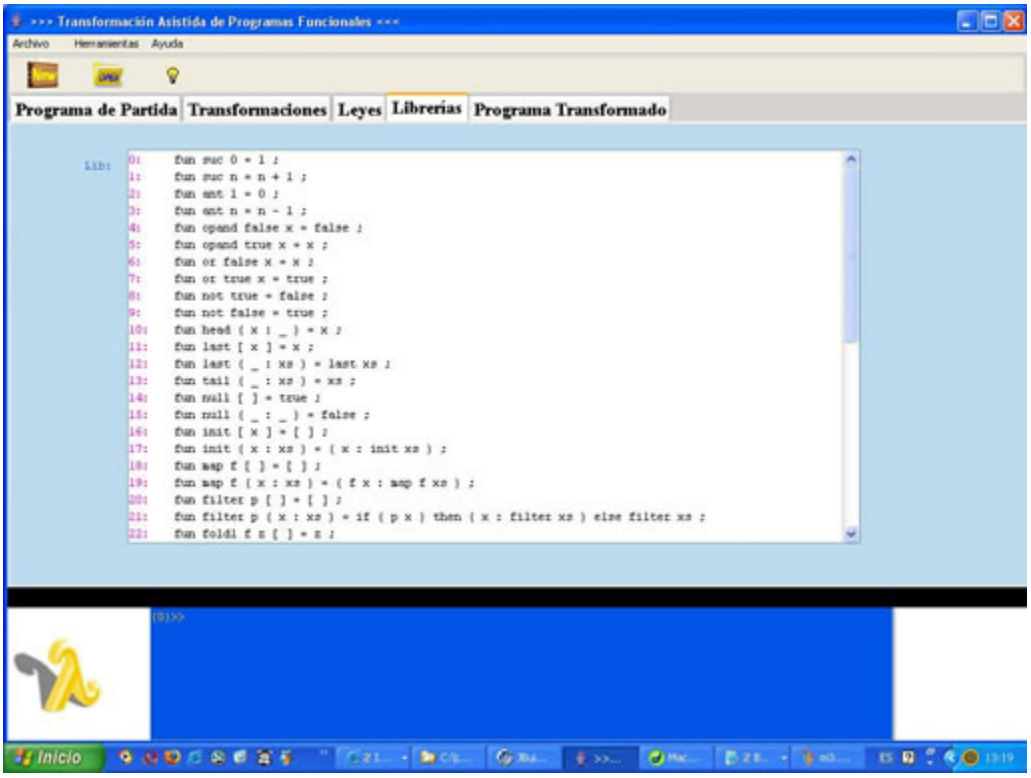
Transformaciones, que es donde está el programa transformado en el momento actual. Hay dos paneles, el superior lleva el programa transformado y el inferior ecuaciones que hayan surgido de hacer, por ejemplo una instanciación, en cuyo caso desaparecen del programa transformado (quedando en el panel de abajo), pero en un futuro se pueden volver a copiar al panel de arriba, volviendo a formar parte del programa transformado. El panel azul de la parte inferior se ve independiente de la pestaña que esté abierta.



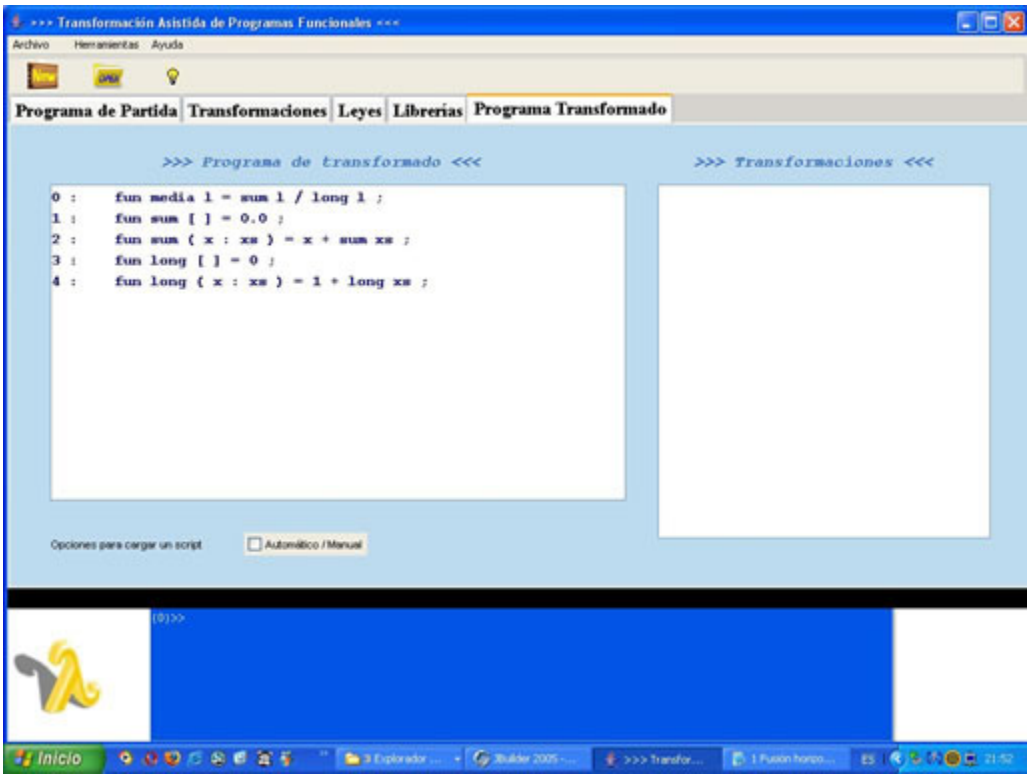
Leyes, lleva la colección de leyes predefinidas y las definidas por el usuario en el transcurso de la transformación actual.



Librerías, que es una colección de las ecuaciones de varios tipos, como enteros, listas, colas, etc.

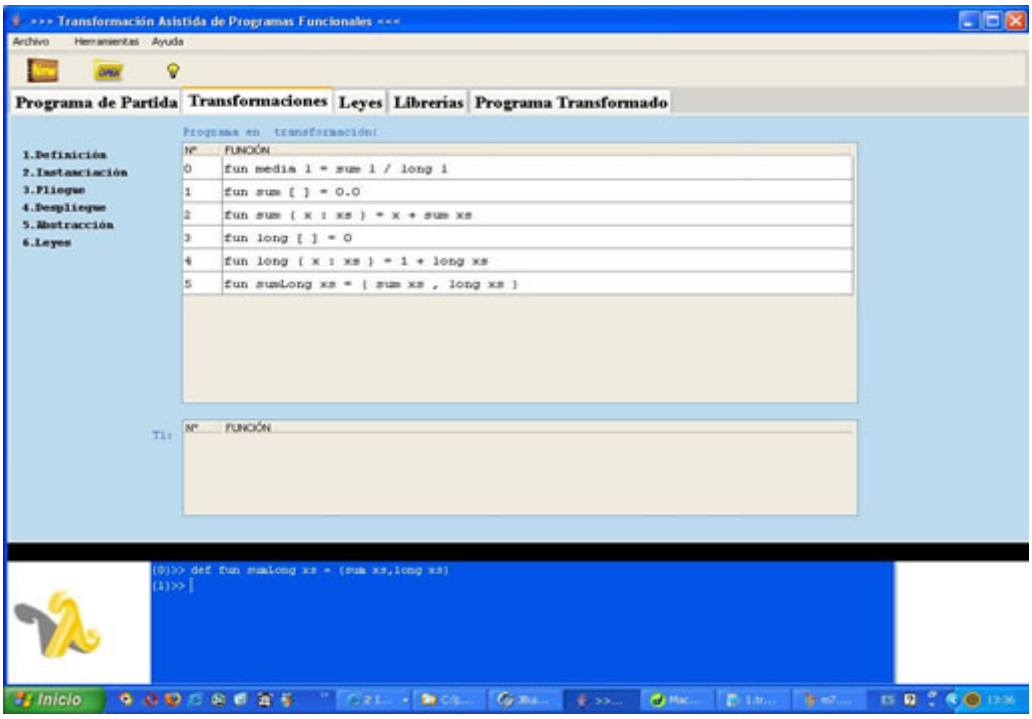


Programa Transformado, lleva el programa transformado en el momento actual, pero con pretty-printer, es decir con tabulaciones. También lleva el historial de las reglas de transformación aplicadas.

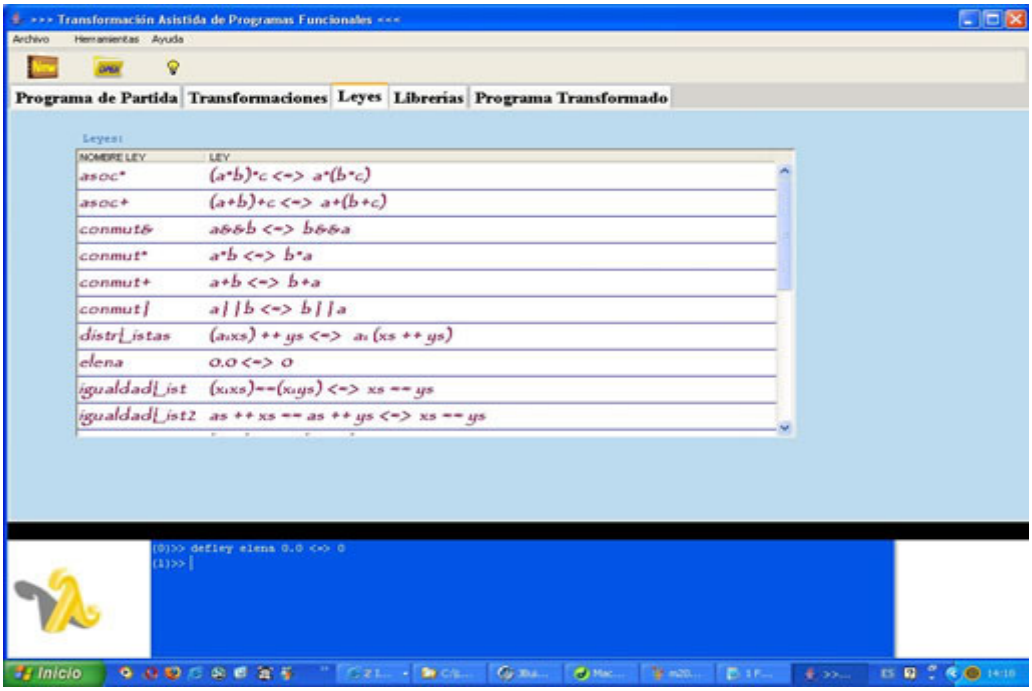


11.4.2. Definición de funciones y leyes

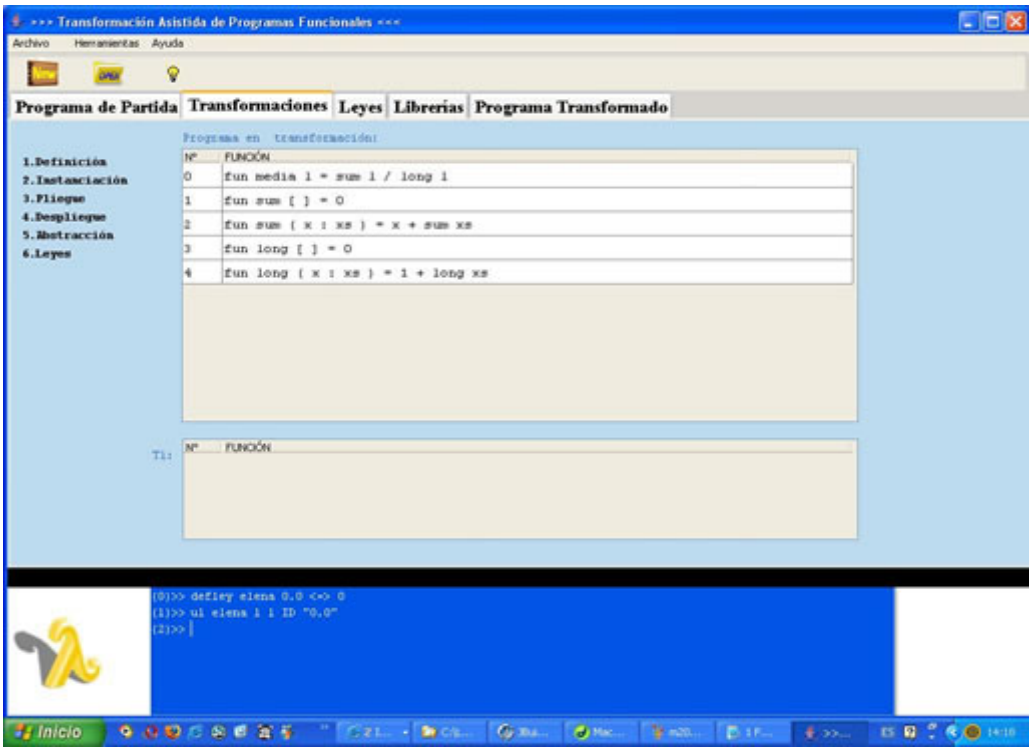
Se puede definir una función, que en un principio no formaba parte del programa de partida, pero que se define como auxiliar y pasa a formar parte del programa transformado. Se define en la consola azul de abajo. La función nueva aparece como la última en el panel de arriba.



La definición de una ley se hace de manera similar, como ejemplo definimos una ley que se llama ***elena***, se ve como aparece entre las leyes predefinidas.

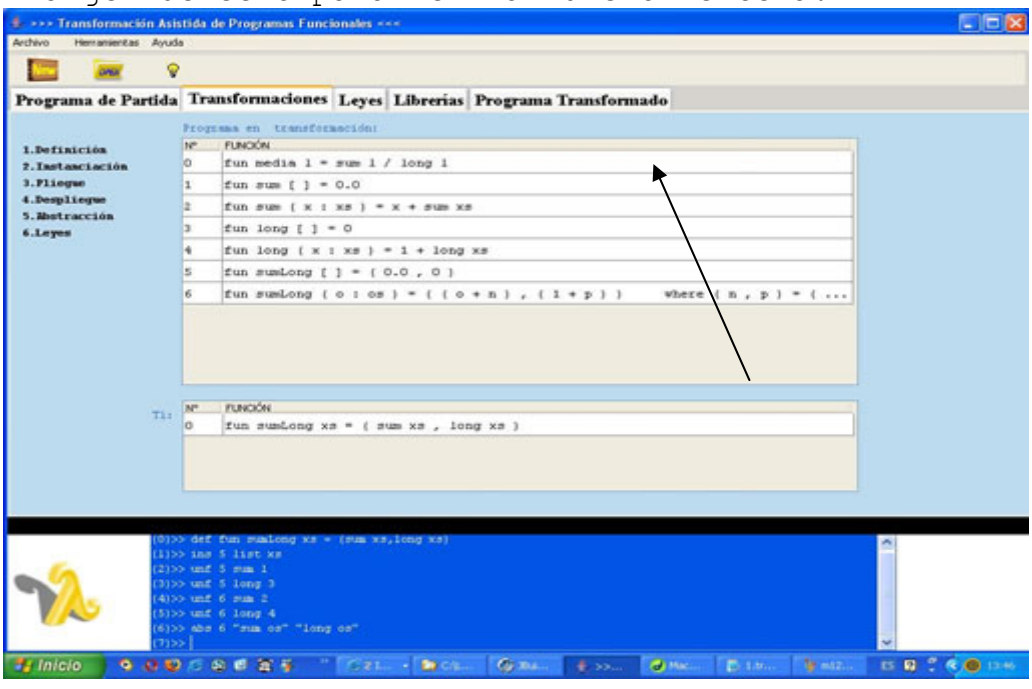


Ahora ya se puede usar la ley *elena*, que lo que hace es cambiar 0.0 por 0 en la línea indicada.

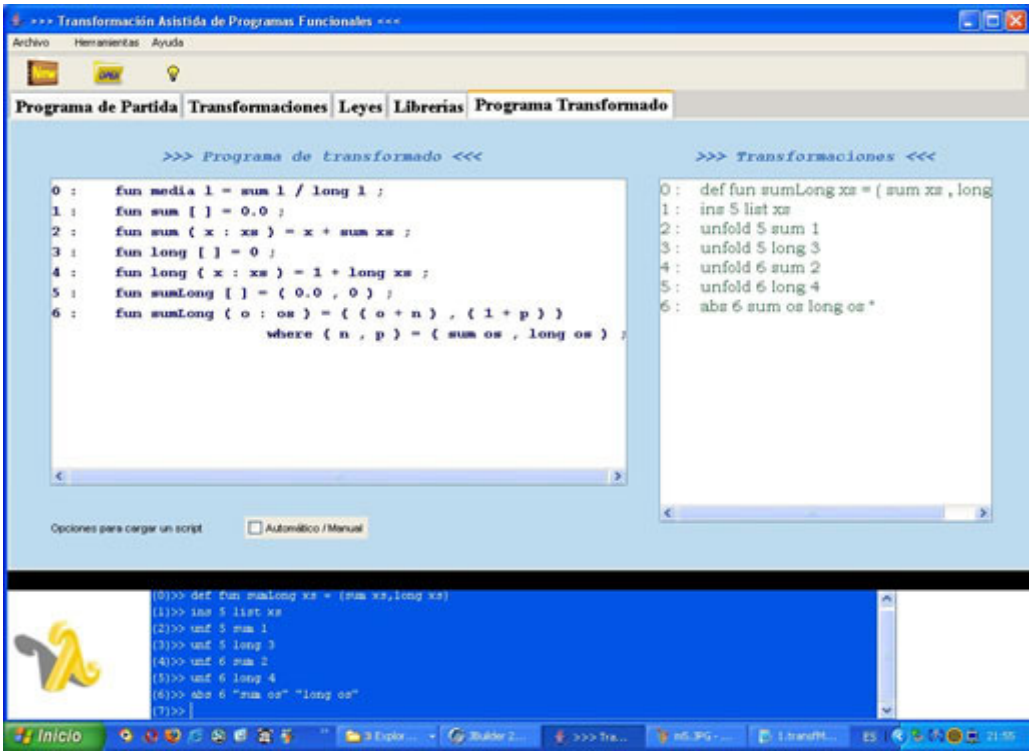


11.4.3. Uso de algunas instrucciones y opciones

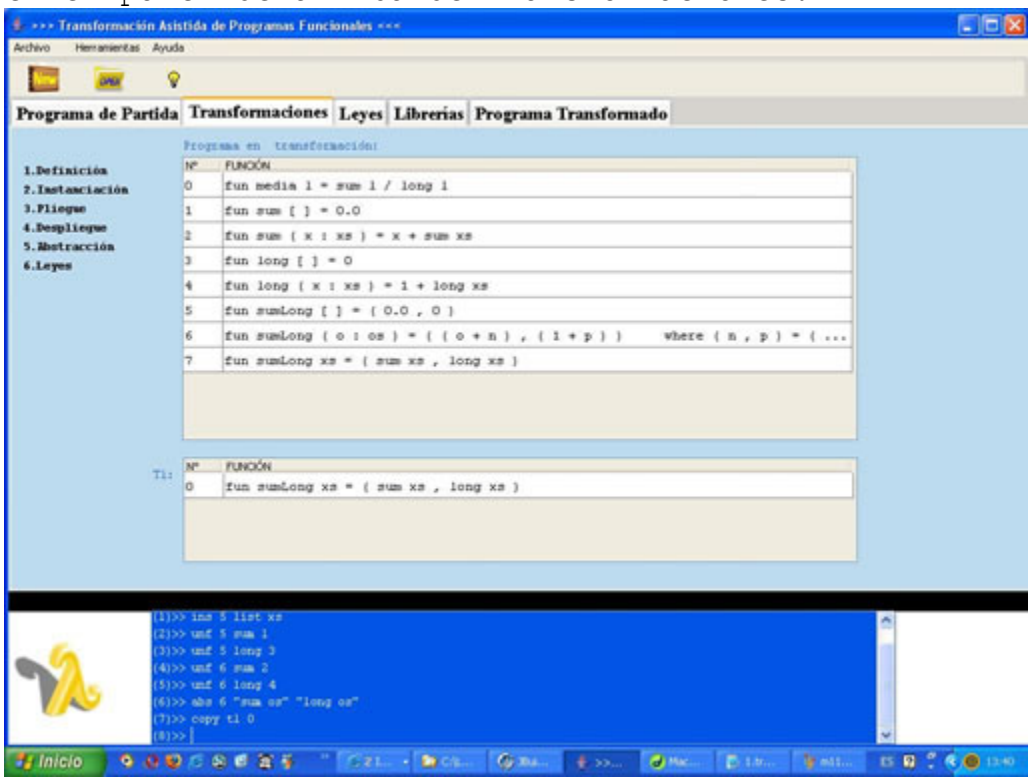
Abstracción: por lo general genera funciones muy largas que tienen *where*. Por eso si lo vemos en la pestaña de Transformaciones, tenemos que ampliar el margen derecho para ver la función entera.



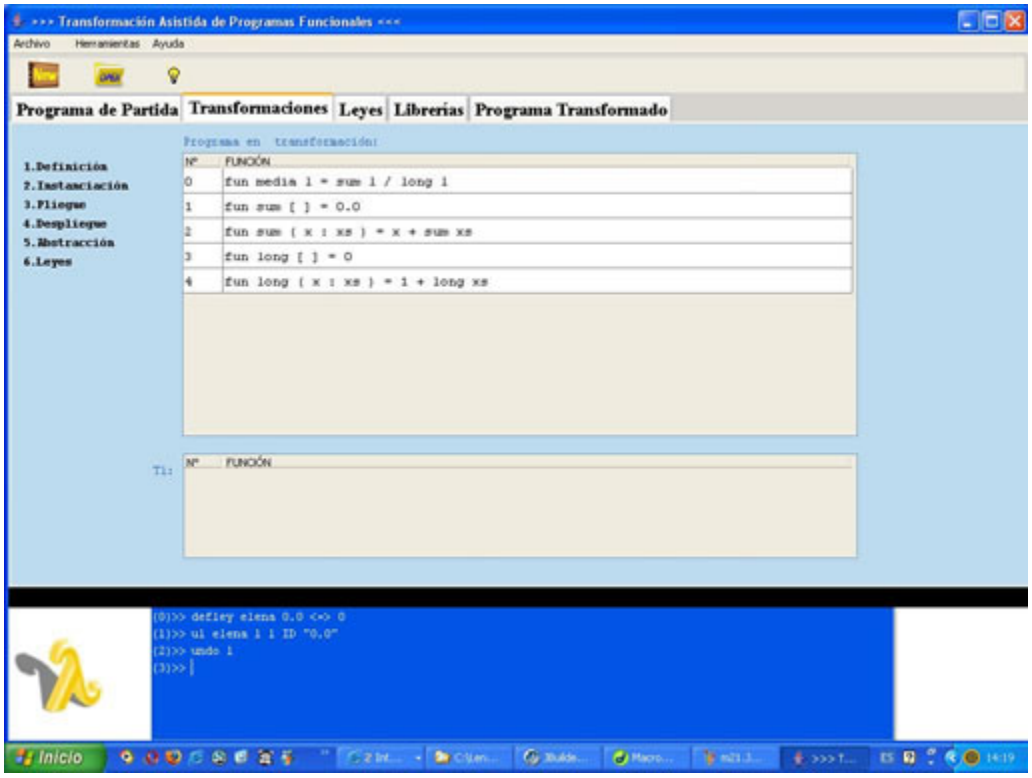
Para ver mejor este tipo de funciones largas se puede ir a la pestaña Programa Transformado y verlo tabulado.



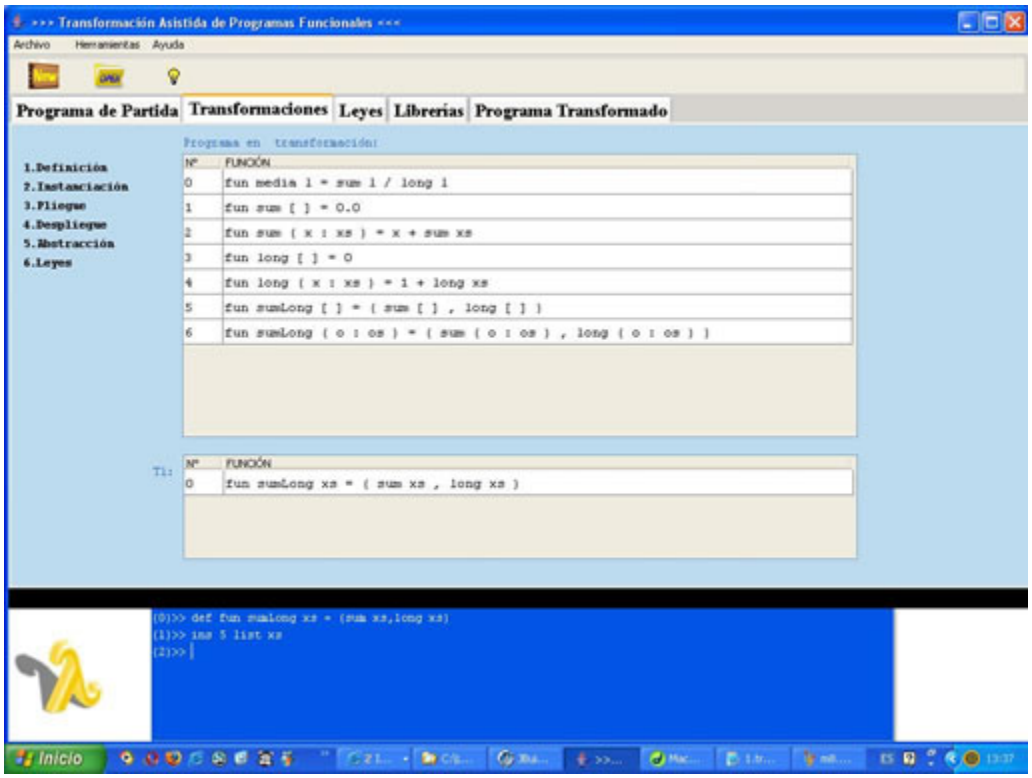
Copiar: hay que indicar si se copia del panel t1 o de las librerías (en este caso vamos a copiar del panel t1), y se ve que esa función aparece la última en el panel de arriba de Transformaciones.



Deshacer: vuelve a hacer todas las instrucciones hasta el paso indicado.

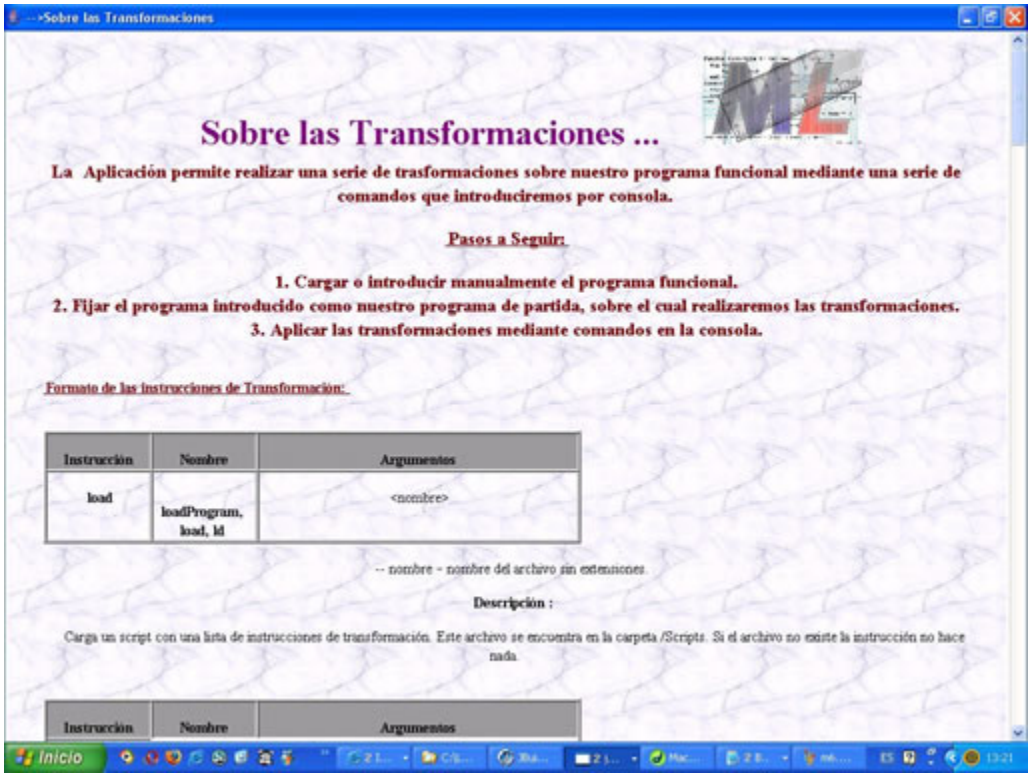


Instanciación: al instanciar una variable en una función se generan dos funciones nuevas que van a formar parte del programa transformado y la función antigua pasa al panel t1 de abajo.

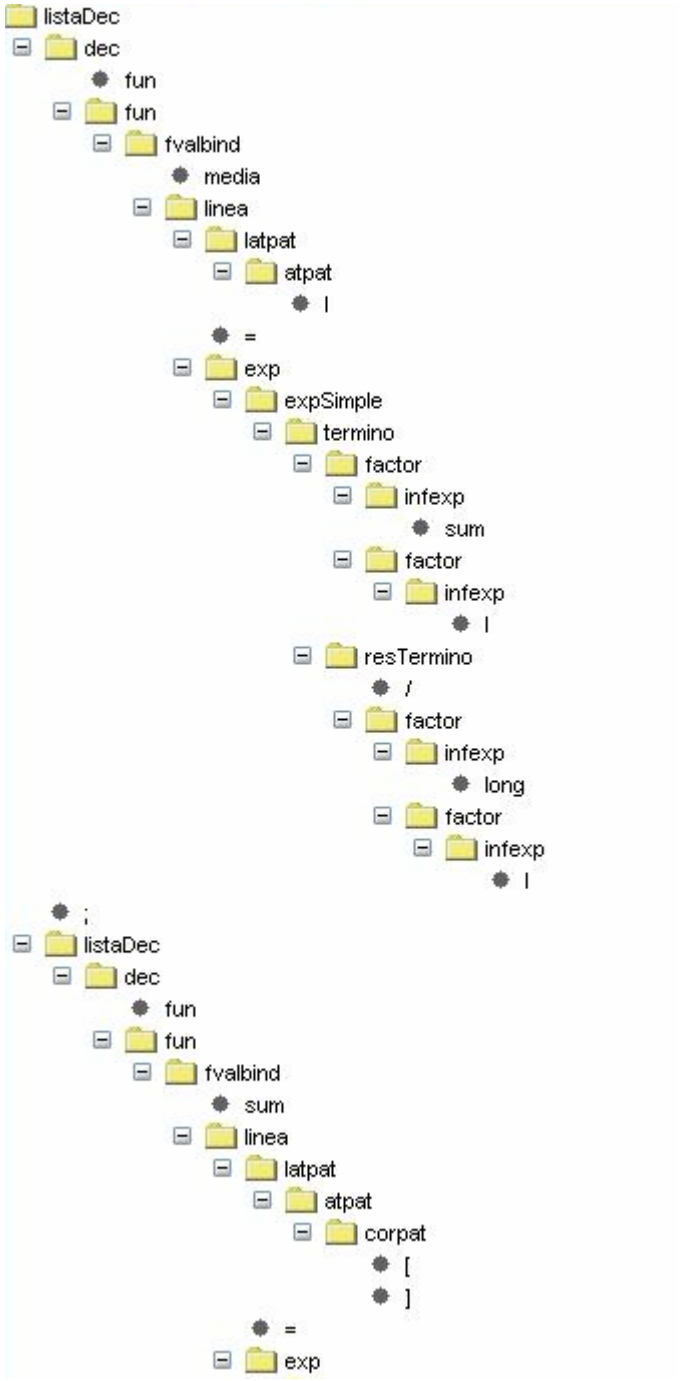


11.4.4. Ayuda

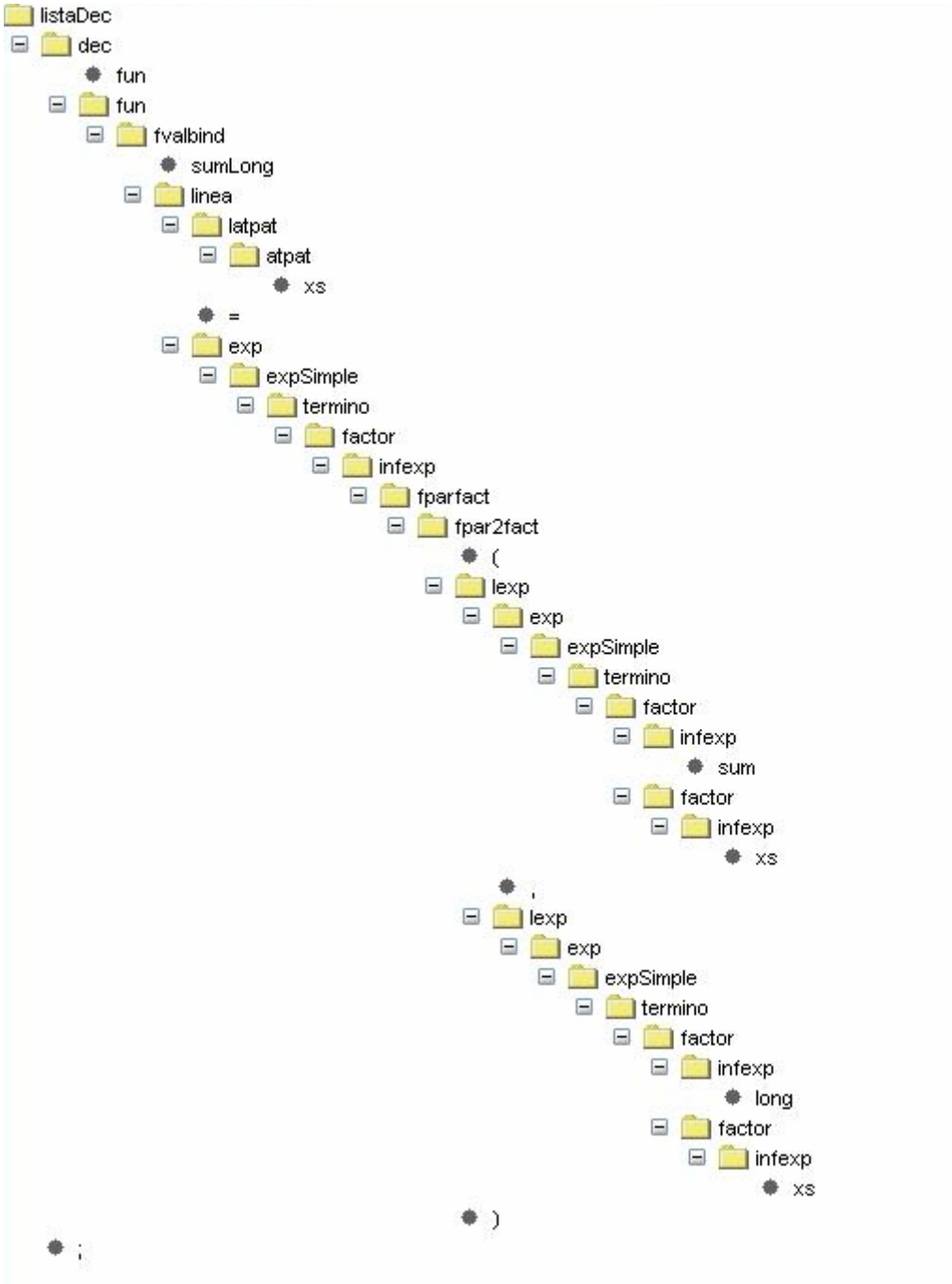
En el menú superior aparecen algunas ventanas de ayuda. Concretamente en *Ayuda -> Documentación del Proyecto* está toda la información referente a los archivos utilizados para programar la aplicación, la estructura de los paquetes, etc. En *Ayuda -> Acerca de…* viene el nombre de los autores de la aplicación. En *Ayuda -> Sobre las Transformaciones* aparece el formato de todas las instrucciones que se pueden aplicar, aquí se ve como es esta última ventana.



En el menú *Herramientas -> Ver Árbol Derivación* aparece el árbol sintáctico del programa transformado:

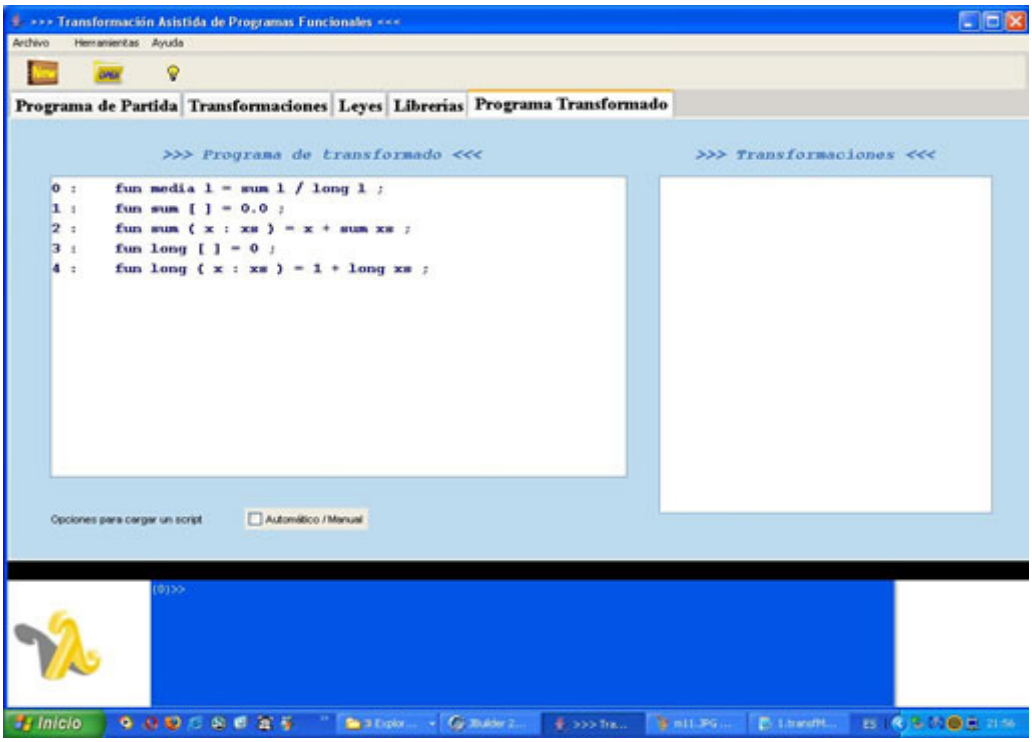


En el menú *Herramientas -> Ver Árbol Derivación2* aparece el árbol sintáctico del panel t1 (el panel de abajo):

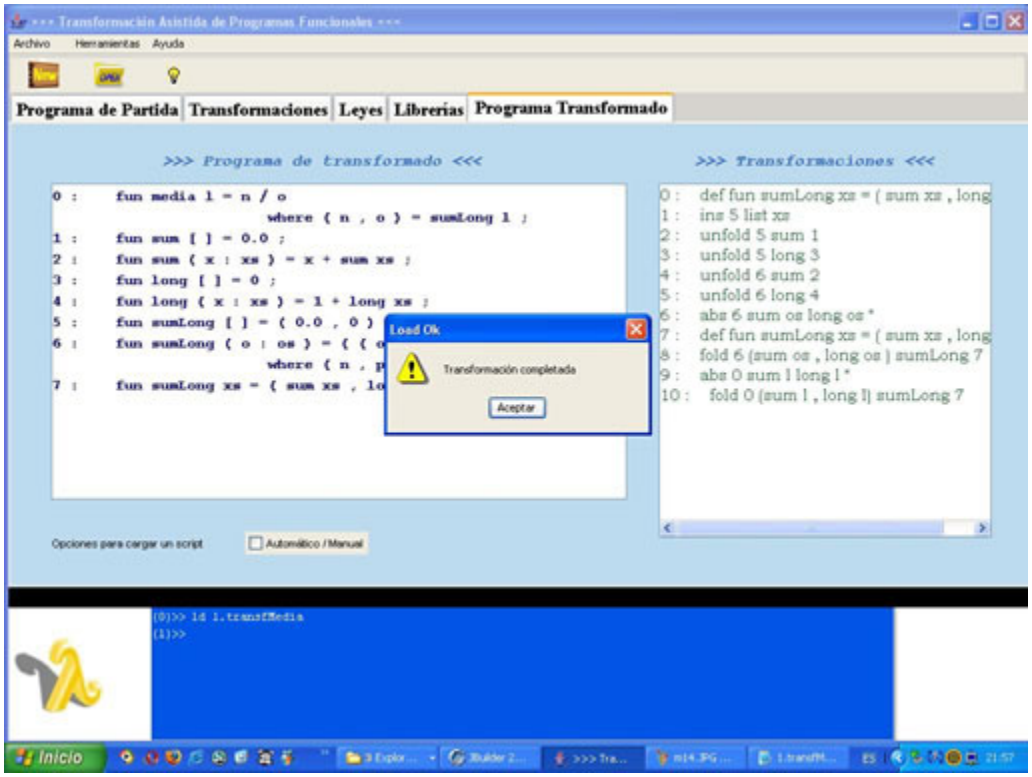


11.4.5. Opciones auxiliares

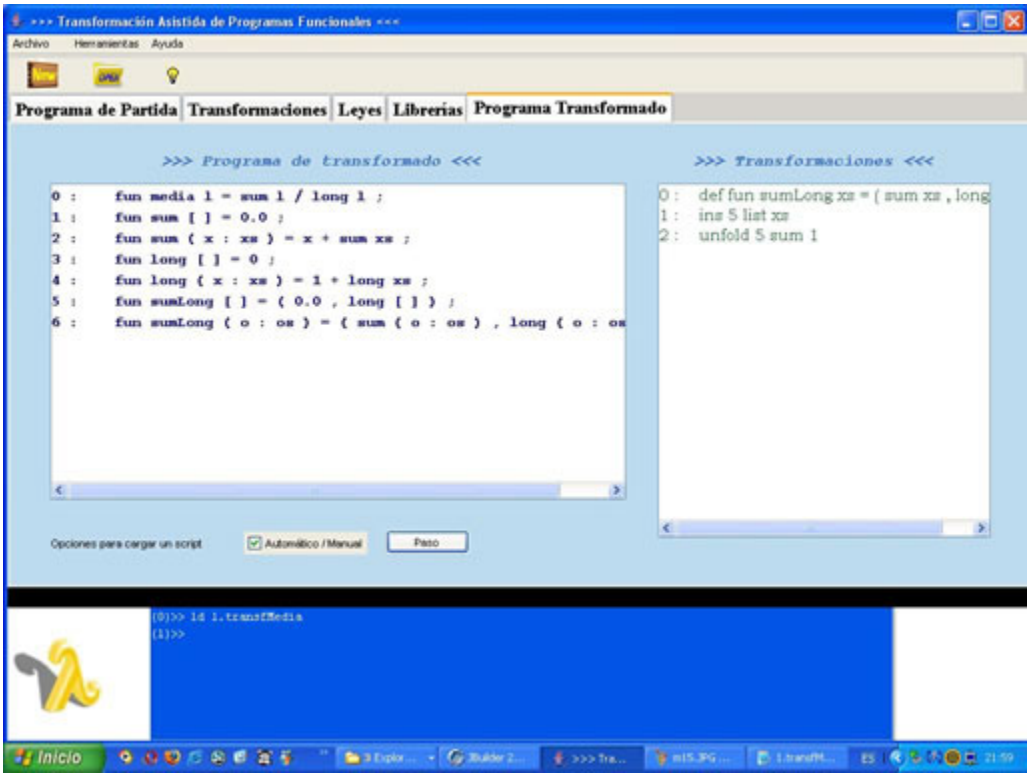
Cargar un *script*: si tenemos un programa inicial como este



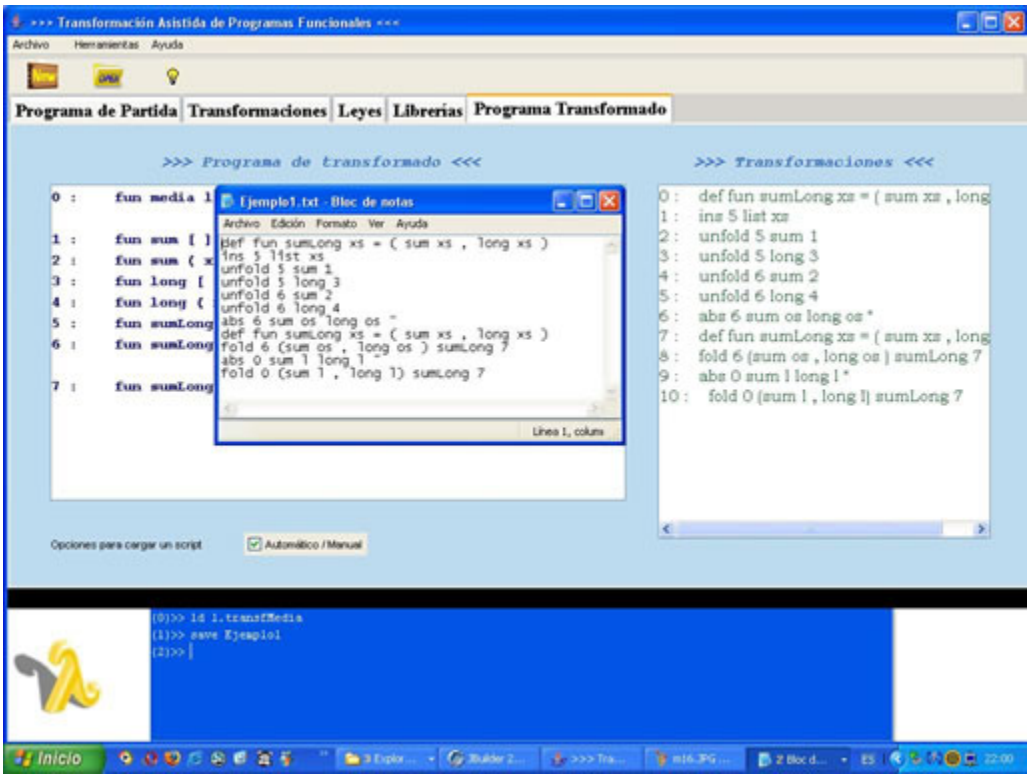
Podemos realizar la ejecución de una transformación automáticamente cargando un *script*:



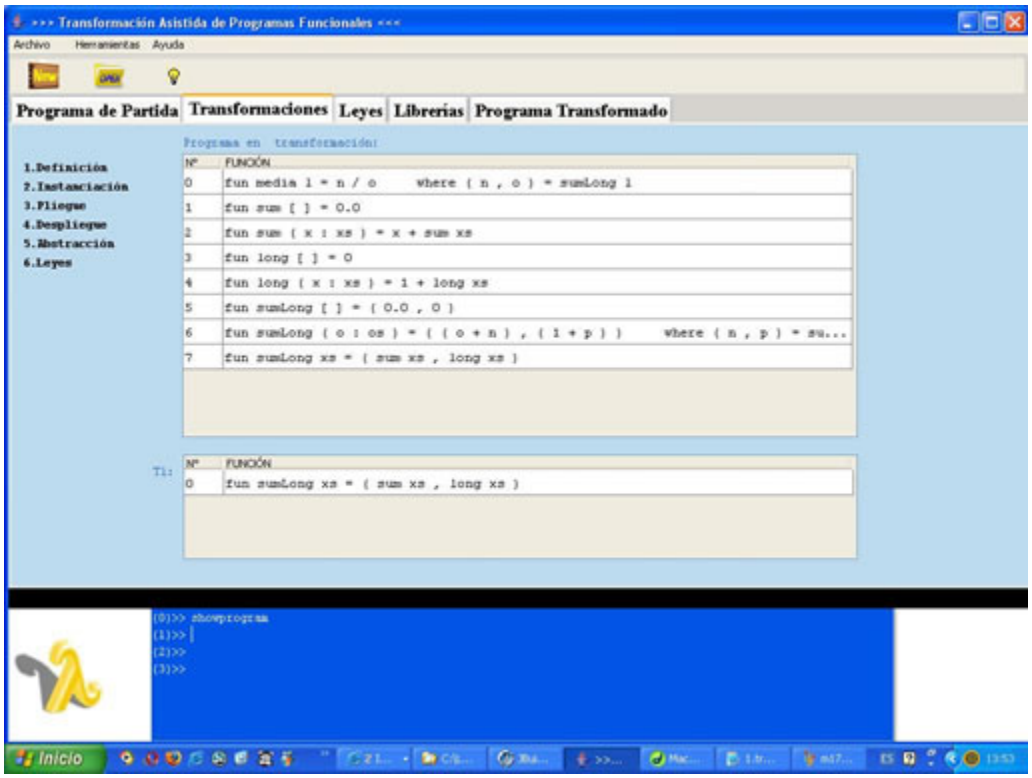
O se puede realizar de forma manual, viendo la transformación paso a paso (dándole al botón Paso).



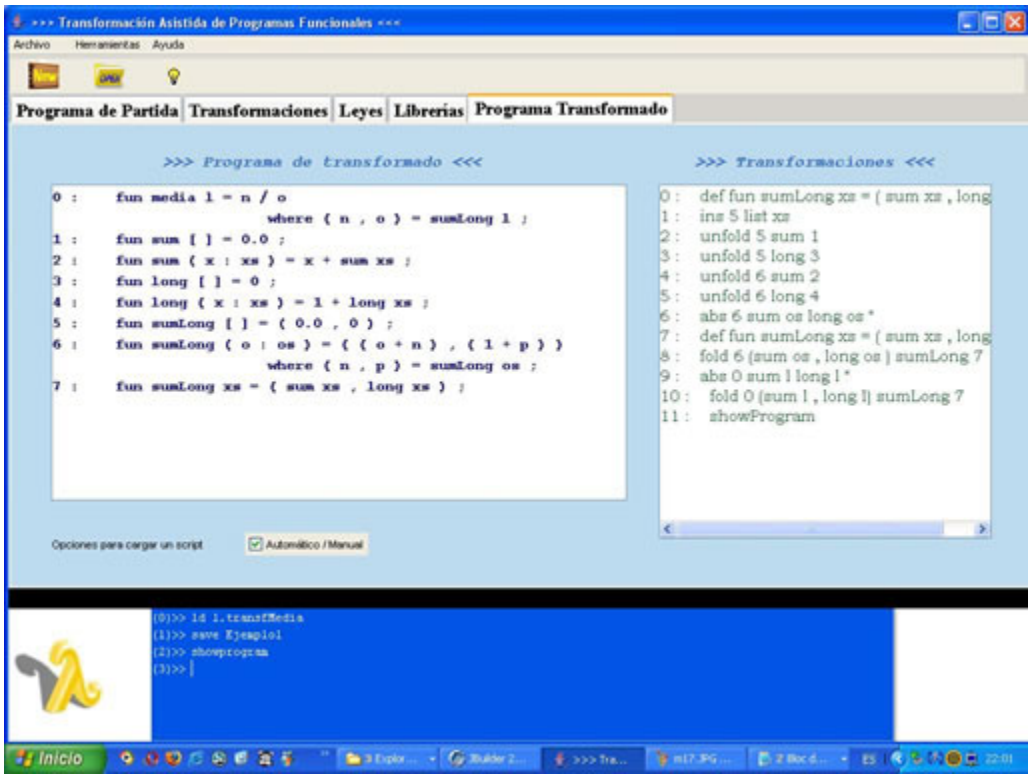
Guardar un programa: se puede guardar la secuencia de reglas de transformación utilizadas en un *script*.



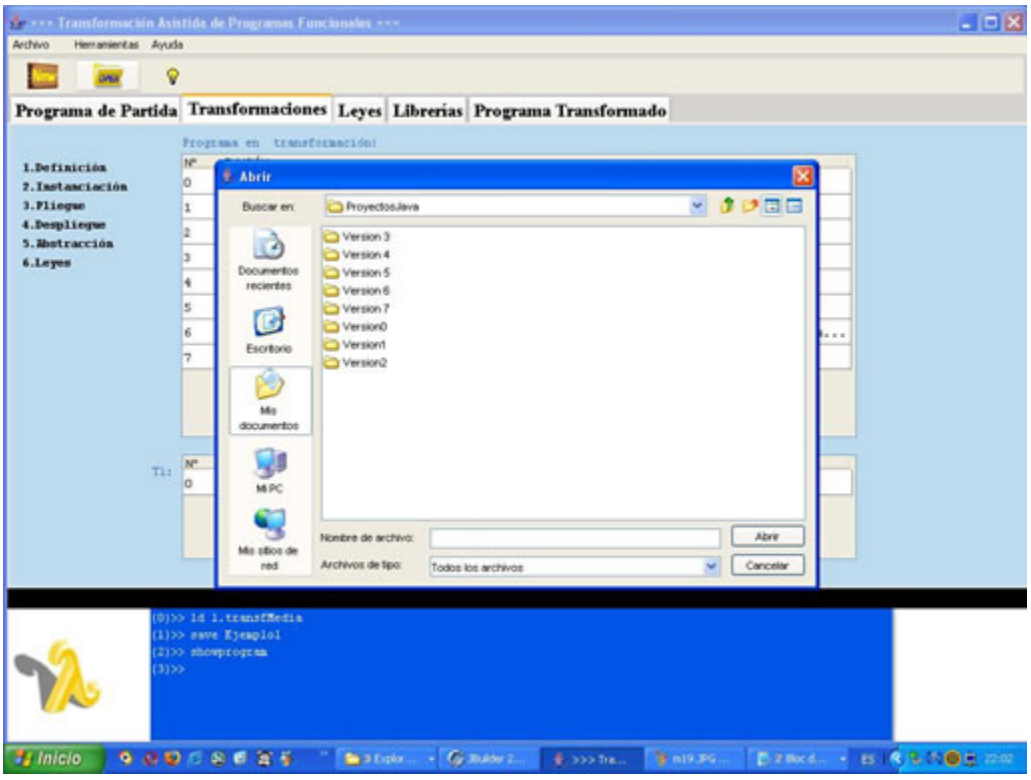
Mostrar reglas de transformación: permite ver todas las reglas de transformación relevantes realizadas hasta el momento (sin load, por ejemplo), es decir si tenemos este programa transformado,



después de showprogram aparece



Abrir: sirve para abrir un programa de partida. Se puede hacer por el menú contextual de arriba, o dándole al botón Open.



12 BIBLIOGRAFÍA

12.1. Libros consultados

APPEL, ANDREW W.
MODERN COMPILER IMPLEMENTATION IN ML
CAMBRIDGE UNIVERSITY PRESS

APPEL, ANDREW W.
MODERN COMPILER IMPLEMENTATION IN JAVA (SECOND EDITION)
CAMBRIDGE UNIVERSITY PRESS

[BD77] R.M. Burstall y J. Darlington. *A transformation system for developing* recursive *programs*. Journal of the ACM, 24(1), 1977.

[TS84] Tamaki, H. and Sato, T., *Unfold/Fold Transformations of logic programs*, 1984.

[B87] Richard S. Bird, *An Introduction to the theory of lists*. Logic of programming and Calculi of discrete design (ed. M. Bray), SpringerVerlag, 1987.

[LPV05] Luis F. Llana Díaz, Cristóbal Pareja Flores y Ángel Velázquez Iturbide. *DESARROLLO DE SOFTWARE CON EL MODELO TRANSFORMACIONAL*. Depto. de Sistemas Informáticos y Programación. Universidad complutense de Madrid, 2005.

12.2. Consultas de Internet

[MAP] **Sistema MAP**

http://www.ercim.org/publication/Ercim_News/enw36/proietti.html

[US] **Sistema Ultra**

http://research.microsoft.com/~schulte/Papers/ToolSupportForTheInteractiveDerivationOfFormally%20Correct%20FunctionalPrograms(JUCS2003).pdf

[HERA] **Sistema HERA**

http://www.cse.ogi.edu/~andy/pub/era-gui-tech.htm

[MAG] **Sistema MAG**

http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/magsystem.html
http://www.program-transformation.org/Transform/MAG

[SML97] Standard ML of New Jersey

http://www.smlnj.org/

[MML] **Moscow ML**

http://www.dina.kvl.dk/~sestoft/mosml.html

[OCAML] **Objective CAML**

http://caml.inria.fr/

[YFPG] **University of York Department of Computer Science York Functional Programming Group**

http://www.cs.york.ac.uk/fp/
http://www.cs.york.ac.uk/fp/transform.html

[MFTP] **Métodos formales de Transformación de Programación**

http://www.dsic.upv.es/users/elp/german/doc/

[EAAS] **Especialización y Análisis Avanzados de Sistemas**

http://www.ecs.soton.ac.uk/~mv/research/asap.php?lang=es#Herme2003

[APT] **Advanced Program Transformation**

http://www.diku.dk/forskning/topps/activities/pgmtrans/

[TPL] **Transformational Programming Literature**

http://citeseer.ist.psu.edu/context/53787/0

[CTC] **Compiladores Traductores y Compiladores con Lex/Yacc, JFlex/Cup y JavaCC**

http://www.lcc.uma.es/~galvez/Compiladores.html

[DSMT] **Desarrollo de Software con el Modelo Transformacional**

http://aljibe.sip.ucm.es/ANIEI