Intérprete de visualizaciones

Proyecto de Sistemas Informáticos

2005-2006

Universidad Complutense de Madrid

Diego Alonso García

Daniel Hernández Arroyo

Carlos Olivar Escaja

Desarrollado bajo la dirección de: **Cristóbal Pareja Flores**

1 Agradecimientos

Como no podía ser menos, queremos dar las gracias a todas las personas que nos han ayudado y apoyado durante los meses que ha durado el proyecto que aquí presentamos.

Estas personas han hecho posible que tanto esmero y dedicación hayan dado su fruto.

A todos vosotros, profesores, padres, hermanos, novias, y amigos….Gracias por todo.

"¿Cuál es la parte más difícil del trabajo de un desarrollador de software?

¿La arquitectura, el análisis funcional, el técnico, la programación?

No. La parte dura de verdad es tener que oír tonterías."

ÍNDICE

1 Agradecimientos 4

2 Autorización 12

3 Resumen 14

3.1 Resumen 14

3.2 Abstract 14

4 Introducción 16

5 Estado del Arte: Visualización de Programas 16

5.1 ¿Por qué es importante la visualización? 16

5.2 Visualización de software 16

5.3 Visualización de software y aprendizaje 17

5.4 Diseño de visualizaciones de algoritmos 18

5.5 Animaciones de algoritmos en la Web 20

5.6 Herramientas de visualización de programas 21

5.7 Conclusiones 23

6 Planteamiento del proyecto 24

6.1 Planteamientos iniciales 24

6.2 Decisiones de diseño preliminares 25

6.3 Decisiones de visualización preliminares 26

6.4 Decisiones de visualización tomadas 28

6.5 Análisis 30

7 Descripción del lenguaje adoptado para el intérprete 34

7.1 Descripción informal del lenguaje fuente 34

7.2 Aspectos léxicos 36

7.3 Aspectos sintácticos: Gramática del lenguaje 37

8 El intérprete 42

8.1 Estructura de un intérprete 43
8.2 Técnicas de interpretación 44
8.3 Reglas para Eval 46
8.4 Reglas para Apply 48
9 Descripción técnica del proyecto 50
9.1 Breve introducción al uso de patrones de diseño 50
9.2 Patrón Modelo-Vista-Controlador 51
9.3 Implementación del código intermedio 53
9.4 Creación de operaciones 53
9.5 Implementación del entorno 55
9.6 Implementación del intérprete 55
9.7 Generación de páginas html 58
10 Conclusiones y trabajos futuros 60
11 Interfaz gráfica y manual de usuario 62
12 Colección de ejemplos 70
13 Bibliografía 90
14 Apéndices 92
13.1 Intérprete Haskell 92

2 Autorización

Los abajo firmantes, Carlos Olivar Escaja, Daniel Hernández Arroyo y Diego Alonso García, autores del proyecto *Intérprete de visualización Cardida* en la asignatura de Sistemas Informáticos, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines exclusivamente académicos, nunca comerciales, y mencionando expresamente a sus autores, los contenidos de este documento, así como el código, prototipos o documentación asociada a dicho proyecto.

Carlos Olivar Escaja Daniel Hernández Arroyo Diego Alonso García

……. …… ……...

Madrid, a…… de julio de 2006

3 Resumen

3.1 Resumen

El objetivo de este proyecto es la implementación de un intérprete de un lenguaje funcional que además añadirá la posibilidad de visualizar el programa que se está interpretando.

La visualización de un programa es la representación gráfica de un aspecto del mismo. Al visualizar su ejecución se mostrarán gráficamente los sucesivos estados por los que pasa el programa. Estos estados vendrán expresados por representaciones gráficas estáticas.

Este proyecto consiste en el diseño e implementación de dicho intérprete de visualizaciones para un lenguaje funcional.

Palabras clave: visualización de programas, visualización de algoritmos, animación de algoritmos, intérprete, programación funcional, proceso de reescritura

3.2 Abstract

The aim of this project is to develop a functional language interpreter that, in addition, will allow the capacity to visualize the program that is being interpreted.

The visualization of a program is the graphic representation of one of its aspects. When visualizing the execution of the program, it will be shown graphically the successive states the program is going through. These states will be expressed by static graphic representations.

This project will suppose the design and implementation of such interpreter.

Key words: program visualization, algorithm visualization, algorithm animation, interpreter, functional programming, rewrite process.

4 Introducción

El objetivo del presente documento es dar a conocer el desarrollo, diseño e implementación llevado a cabo en el proyecto de *Sistemas Informáticos* Intérprete de visualizaciones.

Primero comenzaremos ubicándonos en el mundo de la visualización, y para ello trataremos de dar a conocer en que consiste. Posteriormente, en secciones consecutivas, comentaremos las decisiones tomadas a lo largo del proyecto: planteamiento del mismo, lenguaje adoptado para el intérprete, estructura, implementación, cuestiones de diseño tomadas, etc. terminando con la descripción de una serie de ejemplos predefinidos y que podremos encontrar en un cd adjuntado al presente documento.

5 Estado del Arte: Visualización de Programas

Esta sección pretende ser un repaso al estado actual de la técnica de visualización de programas funcionales. Con ella queremos enfatizar la importancia de una correcta visualización, que justifica los objetivos de nuestro proyecto.

5.1 ¿Por qué es importante la visualización?

La importancia de la visualización radica en su fuerza expresiva. Suele decirse que una imagen vale más que mil palabras. Las características de las imágenes hacen que la búsqueda de la información se reduzca significativamente, a pesar de que en una sola imagen puede aparecer resumida información equivalente a enormes cantidades de datos numéricos. Este alto poder de abstracción es otro punto a favor en el uso de imágenes.

El término *visualizar* significa hacer visible a simple vista lo que no lo es. Hacer visible es representar las nociones abstractas o conceptos que se desean transmitir al receptor de la información. Mediante visualización podemos ayudarnos a la comprensión, diseño, análisis o uso de un concepto o artefacto. Sólo es necesario hacerlo visible mediante una cierta representación.

Los algoritmos y el software no son ajenos a una posible representación visual. Mediante una correcta visualización podemos dar una representación a cosas que carecen de ella, como el software, haciendo evidente el significado de entidades abstractas relacionadas con él. También podemos emplear la visualización para mostrar el funcionamiento de algoritmos. La búsqueda de mejores representaciones visuales para el software ha dado lugar a la investigación sobre visualización de software.

5.2 Visualización de software

La visualización de programas tiene dos vertientes principales. La primera de ellas sería la denominada *programación visual*, que es una disciplina centrada en conseguir especificar más fácilmente los programas por medio de una notación gráfica. Un ejemplo

de programación visual es la notación **UML**. La otra rama de la visualización sería la *visualización de software*, que tiene por objetivo hacer que determinados aspectos de programas y algoritmos se entiendan más fácilmente.

Visualizar software es hacer que un programa tenga un aspecto visible diferente al que tiene su código fuente. El formateo, o el *pretty-printting* de un programa, son formas rudimentarias de visualización, basadas principalmente en los criterios estéticos del programador.

La visualización de la ejecución de un programa podría realizarse desde múltiples niveles de abstracción: desde el nivel de algoritmos, del comportamiento de programas, e incluso de los cambios ocurridos a nivel de hardware. En relación con la ejecución de un programa, los criterios de visualización podrían variar considerablemente según la necesidad de visualizar los cambios de estado y pasos de la reescritura. El programador podría optar por la representación de los aspectos estáticos (dependencias entre objetos, visibilidad…) o dinámicos (ejecución, sustituciones, mostrar cambios de estado…).

Esta necesidad de definir la visualización, especialmente en el caso de la visualización de algoritmos, donde el diseñador puede recurrir a una visualización *ad hoc*, es una de las principales dificultades en la visualización de software.

Además de esta necesidad de definir los eventos significativos para la visualización, es necesario registrarlos de alguna manera. Establecer los mecanismos necesarios para obtener esa información de la ejecución de un programa se denomina "instrumentación" de los programas. Idealmente la instrumentación de un programa no debería modificar el comportamiento del mismo.

Los sistemas de visualización de software se emplean fundamentalmente, y con resultados bastante positivos, con fines pedagógicos. Como se señala en [Enc00], la comprensión de los lenguajes funcionales y su depuración puede no ser sencilla, al involucrar aspectos que pueden ser difíciles de entender, como la pereza, o que pueden resultar más intrincados y engañosos de lo que parecen a primera vista, como el orden de las declaraciones. La facilidad a la hora de transmitir o mostrar algunos de estos conceptos es sin duda uno de los mayores beneficios de la representación de software.

5.3 Visualización de software y aprendizaje

Se ha propuesto la visualización y animación de programas, entre otras muchas aplicaciones de esta disciplina (como la depuración de programas funcionales), como un método para mejorar la comprensión de programas entre alumnos principiantes.

Las razones para suponer que la visualización de un programa puede mejorar el aprendizaje radican en el uso de la visualización como una forma de construir modelos mentales de fenómenos abstractos.

Un modelo mental es una estructura cognitiva que la mente humana usa para representar conocimiento. Si aceptamos que el conocimiento humano no es enteramente verbal, sino visual, entonces las experiencias visuales deberían ser eficaces a la hora de estimular el aprendizaje. Además, muchos fenómenos no son sólo visuales, sino que además son dinámicos, lo que ocurre con frecuencia en programación. La animación, junto

con la visualización, debería ser también una ayuda eficaz para comprenderlos, que puede justificar el uso de esta disciplina en la docencia.

En [Ben01] podemos ver las principales conclusiones de la investigación llevada a cabo por Mordechai Ben-Ari en el Department of Science Teaching del Weizmann Institute of Science (Rehovoth, Israel) sobre un curso anual de introducción a la informática.

En dicho curso, se usó una adaptación de Jeliot, un sistema de animación de programas que usa un subconjunto de la versión 1.4 de Java y puede consultarse en [JE06]. Dicha herramienta, centrada en la animación de programas, proporcionaba interfaces de usuario demasiado complejas para estudiantes principiantes, por lo que fue adaptado por Ben-Ari para su uso con animaciones y etiquetas de cada aspecto de la ejecución de programas sencillos.

La comparación entre los alumnos de un grupo que había recibido formación suplementaria con Jeliot para el curso de introducción a la informática y los que no, reveló que el uso de animaciones y visualizaciones otorgaba un vocabulario de términos más amplio y mejor para las explicaciones y las predicciones en el funcionamiento del programa a los alumnos que había recibido formación con la herramienta de visualización. Además, imitar la animación ayudaba a los alumnos a resolver problemas nuevos.

Sin embargo Ben-Ari concluía su artículo recalcando que la animación no es la solución definitiva en enseñanza. El simple hecho de visualizar algo no conduce necesariamente a mejoras en su aprendizaje. Las representaciones gráficas sólo son útiles bajo ciertas condiciones.

La visualización debe ser diseñada para una clase concreta de estudiante (en el caso de la investigación de Ben-Ari, para estudiantes novatos). La presentación de gráficos y texto deben ir unidos y mostrarse de forma simultánea y coordinada. La visualización puede aumentar la motivación del usuario, pero también es necesario invertir tiempo no sólo en diseñar la visualización apropiada, sino en enseñar al usuario a ver lo que la visualización le muestra y a trabajar con ella.

Por tanto, podemos destacar de esta investigación que, usada con cuidado y en las condiciones adecuadas, la visualización puede ser una herramienta importante para la docencia.

5.4 Diseño de visualizaciones de algoritmos

Los avances en la tecnología, tanto a nivel hardware como software, hacen posible hoy en día ejecutar visualizaciones computacionalmente exigentes con simples ordenadores personales. Sin embargo, como se ha mencionado previamente, es necesario prestar atención al análisis de los usuarios, de sus necesidades, tareas y objetivos para dar lugar a una representación adecuada de la información que sea útil para esos usuarios.

Durante largo tiempo, se ha usado la visualización para la enseñanza en informática. En concreto, para el diseño y análisis de algoritmos, produciendo representaciones gráficas de programas para facilitar su depuración y documentación.

La creación de visualizaciones educativas de algoritmos supone un tiempo y esfuerzo considerables, ya que el paso de un algoritmo a una representación animada no es un problema trivial.

El problema es que las visualizaciones se desarrollan para atender a usuarios con conocimientos dispares y necesidades complejas y específicas. Los sistemas de visualización, como deben tener muy en cuenta el tipo de usuarios a quienes van dirigidos y las necesidades y conocimientos de estos para que la visualización tenga un impacto positivo en la enseñanza de algoritmia, requieren un esfuerzo notable de análisis de la especificación y diseño de la representación.

Por este motivo, ningún sistema de visualización de algoritmos podrá ser el mejor para todos los usuarios y tareas al mismo tiempo. En la visualización de algoritmos es necesaria la especialización. Esto es un problema añadido al esfuerzo de análisis y diseño, ya que el producto final no podrá ser tan ampliamente distribuido, como para cubrir las necesidades de grandes grupos de estudiantes de programación.

Comprender quiénes son los usuarios del producto permite determinar mejor los contenidos de organización, anchura, profundidad y métodos de presentación de la información. Se trata de analizar las necesidades y conocimientos de estos usuarios, lo que normalmente va mucho más allá de distinguir usuarios novatos de experimentados en el uso y manejo de este tipo de herramientas. Sin embargo, hay un conjunto de necesidades que resultan básicas para todos los tipos de usuarios[1]:

- Medios gráficos consistentes para controlar la propia herramienta y la visualización del programa o algoritmo que se está representando, tales como botones, menús…

- Una estructura homogénea en la definición de las visualizaciones, que facilite la comprensión de ejemplos variados.

- Una colección de ejemplos predefinidos, y además la capacidad gráfica para trabajar con problemas nuevos.

- Indicaciones claras sobre el manejo de la herramienta, así como de los pasos del algoritmo durante su ejecución.

- Capacidad para controlar la velocidad con la que avanza la evolución del algoritmo.

Además de estas necesidades comunes y de las específicas mencionadas anteriormente, para controlar el aprovechamiento de las ventajas de la visualización de algoritmos, es necesario realizar un seguimiento de la evolución de los alumnos que usan dicho material. A esto hemos de añadir que en un sistema diseñado para enseñar en clase se debería optar deliberadamente por mostrar sólo las representaciones correspondientes a los aspectos algorítmicos, ocultando los detalles de la programación e implementación de los mismos para alejarlos de la mente de los alumnos y no entorpecer el aprendizaje del algoritmo.

[1] En la sección *5.6 Herramientas de visualización de programas* podremos ver en que medida, nuestro intérprete de visualización *Cardida* las cumple.

Para que una visualización de un algoritmo sea eficaz, dicha visualización debería demostrar gráficamente los efectos de los algoritmos sobre las estructuras de datos. La pregunta clave en la visualización de algoritmos sería: "¿Necesitan los usuarios esta información presentada de esta forma?".

La representación gráfica de la información depende fuertemente del concepto mostrado. Para datos de gran tamaño, o para estructuras complejas, se necesita mostrar cómo cambia la relación entre los objetos a medida que avanza en el tiempo la ejecución del algoritmo. Por lo tanto se puede concluir que la animación debería ser la elección correcta para la representación de algoritmos.

Además, para representar información se puede usar la forma, el tamaño, el color y otros efectos más sutiles como la textura, la disposición de los objetos, sonidos y efectos en tres dimensiones. Cabe resaltar que si bien estos elementos pueden usarse para mejorar la representación, conviene no abusar de ellos. Por ejemplo, el color puede llamar la atención sobre datos específicos, o identificar elementos y estructuras, pero demostraciones prácticas aconsejan no usar más de cuatro colores en las representaciones gráficas dirigidas a alumnos principiantes, ya que lejos de centrar la atención, se produce el efecto contrario.

Por último, comentaremos los dos enfoques seguidos en la visualización de algoritmos. El primer enfoque es el de la *visualización de la presentación unificada*. Este enfoque consiste en usar una representación gráfica para distintos algoritmos de la misma clase. La representación fijada para el primer algoritmo se puede usar para los demás, lo que ahorra tiempo. Además, al tener una base común, el comportamiento de los distintos algoritmos se compara fácilmente.

El segundo enfoque es usar una *representación única para cada algoritmo*. Este enfoque busca la mejor representación para cada algoritmo, lo que consume bastante tiempo y esfuerzo. Además, la comparación de algoritmos cuesta más trabajo. A cambio, se puede enfatizar en los detalles particulares de cada algoritmo.

5.5 Animaciones de algoritmos en la Web

Tradicionalmente, la visualización de software ha sido una disciplina relegada al ámbito científico y al uso pedagógico. El desarrollo de sistemas de animación para uso educativo ha dado lugar a muchas páginas Web con colecciones de animaciones.

Las animaciones presentan una serie de problemas prácticos a la hora de aplicar su uso a la enseñanza. Como se ha mencionado anteriormente, las animaciones no garantizan por sí solas el éxito pedagógico de la herramienta que las usa. Deben cumplirse ciertas condiciones, tanto entre el alumnado como en las animaciones. Además, debe añadirse la dificultad de generar una animación correcta.

En [Nov01], Cristóbal Pareja, J. A. Velázquez, Fernando Naharro y Margarita Martínez Santamarta, presentan un artículo de investigación referente a la publicación de animaciones de algoritmos en la Web. En el citado artículo se enumeran una serie de características básicas para la eficacia educativa de las animaciones de algoritmos. Dichas características son:

- Integración de explicaciones en las animaciones de los algoritmos.

- La posibilidad de avanzar y retroceder en los pasos de la animación.

- Involucrar activamente al usuario en la animación.

Los autores proponen la generación automática de páginas Web con las secciones de descripción del problema, descripción del algoritmo, texto del programa y animación del algoritmo. Se usan índices locales para el acceso rápido a las distintas secciones e interfaces de control para la animación, que muestra dos fotogramas a la vez para indicar explícitamente las diferencias de un estado y otro. La motivación es que dos fotogramas consecutivos pueden resultar más útiles a un estudiante a la hora de entender la transición de un estado a otro.

Para dicha investigación se extendió el entorno de programación **WinHIPE** para generar páginas Web con animaciones. Se añadió al programa de la capacidad de generar plantillas en HTML y luego se completaba la información específica de la animación. De esta forma se le dotó de la capacidad para generar páginas Web que incluían animaciones de algoritmos. El objetivo era el uso de animaciones para una asignatura de estructuras de datos y programación en el paradigma funcional.

Dicha investigación prosigue en la actualidad, y en parte ha motivado este intérprete de visualización. Futuros planteamientos de la investigación incluyen medir la eficacia educativa de herramientas que permitan distintos tipos de interacción con programas y algoritmos, con el fin de mejorar la enseñanza de programación funcional y algoritmia.

5.6 Herramientas de visualización de programas

Existen muchas herramientas de visualización de software en la Web, por lo que puede resultar interesante compararlas para examinar el estado actual de la visualización de software, lo que ya hay hecho y las novedades que se podrían aportar para eliminar las carencias de esta disciplina y obtener mejores resultados en las áreas donde se apliquen.

Dada la naturaleza de este proyecto y sus intereses, nosotros centraremos la búsqueda sobre herramientas que visualizan programas funcionales o realizan animaciones de algoritmos.

Ya se ha dicho anteriormente que existen un gran número de herramientas para visualización. El problema es que la mayoría de estas herramientas sólo tratan áreas específicas de la aplicación, por lo que a veces los usuarios deben usar varias herramientas distintas según intereses específicos. Por supuesto, las herramientas especializadas ofrecen mayores y mejores prestaciones. Sin embargo el precio de esa especialización suele ser una limitación en otras áreas que pueden resultar importantes para una correcta visualización.

Por ejemplo, sistemas que no pueden deducir correctamente la representación más adecuada para una estructura de datos de tipo lista, harían que una pila implementada con una lista enlazada se mostrase como una lista, cuando el objetivo, al menos desde un punto de vista pedagógico, sería que fuese mostrada de la forma más parecida posible a una pila lógica.

No olvidemos las previamente citadas características que debe tener una buena aplicación de visualización de software. Con herramientas especializadas nos encontramos

normalmente con la obligación de elegir entre la automatización de la generación de imágenes y el control absoluto sobre todos los elementos a mostrar en la representación. Nuestras necesidades nos llevan a buscar una herramienta lo bastante genérica como para mostrar adecuadamente la presentación al tiempo que manejamos el contenido a presentar.

De las herramientas examinadas en [Enc00] durante nuestra documentación del tema, podemos destacar algunas como Buddha, Freja o **Hat**, que trabajan sobre subconjuntos de Haskell. De estas, sólo Hat (Haskell Tracing System) [Hat] tiene un interfaz gráfico. Existen otras herramientas como **Hood** [Hoo] y GHood que trabajan sobre Haskell completo y sin problemas de memoria, pero de ellas sólo GHood (puede consultarse [Gho01] para más información) tiene interfaz gráfico.

El examen de estas herramientas nos lleva a la conclusión de que existe un amplio abanico para depuradores algorítmicos para Haskell (Buddha, Freja) y depuradores de visualización de datos (como GHood).

Sin embargo, las características de nuestro proyecto nos encaminan más hacia herramientas como **WinHIPE** [Win]. En [JVI] podemos ver el análisis que se hace de la herramienta desde el punto de vista de la interacción del usuario con la animación. La principal baza de WinHIPE sería su capacidad de representación automática de tipos de datos algorítmicos (árboles, listas…) para su visualización en entornos Web. El único punto en contra sería que WinHIPE visualiza Hope; a pesar de ser un lenguaje funcional, Hope está algo obsoleto en el mundo de la programación declarativa.

Nuestra herramienta **Cardida** cumple con ciertas necesidades básicas para los usuarios: posee una interfaz gráfica simple y consistente (permite un fácil manejo de la herramienta), dispone de una colección de ejemplos predefinidos, visualizaciones homogéneas y capacidad de generar *html* que integra secciones de descripción del problema, descripción del algoritmo, texto del programa, además de la propia animación. Por otro lado posee carencias y limitaciones que podremos ver en la tabla comparativa, además de otras como no poseer capacidad de controlar el proceso de evaluación o cambiar focos de atención en la visualización.

A continuación mostramos una tabla comparativa de las distintas herramientas examinadas:

	Compartición	Haskell completo	Modificación de código	Interacción con el usuario	Modificación del orden de evaluación	Compiladores	Problemas de memoria	Impaciente/ perezosa	Interfaz gráfico
Hood	No	Sí	Sí	No	No	ghc	No	Perezoso	No
Buddha	Sí	No	No	Sí	No	ghc	Sí	Perezoso	No
Freja	Sí	No	No	Sí	No	¿?	Sí	Perezoso	No
Hat	Sí	No	No	Sí	Si	Ghc nhc 98	Sí	Perezoso	Sí
Ghood	No	Sí	Sí	Sí	No	ghc	No	Perezoso	Sí
Prospero	Sí	Miranda	No	Sí	No	¿?	Sí	Impaciente	Sí
Hiot	Sí	No	No	Sí	No	¿?	Sí	Perezoso	Sí
Cider	Sí	Curry	No	Sí	No	Cider	Sí	Perezoso	Sí
WinHipe	Sí	Hipe	No	Sí	No	WinHipe	No	Impaciente	Sí
Cardida	No	Cardida	No	Sí	No	¿?	No	Impaciente	Sí

Tabla comparativa

5.7 Conclusiones

La visualización de software y algoritmos es una línea de investigación que tiene prometedoras posibilidades para la docencia. Acompañar las explicaciones de los entresijos de la programación de lenguajes funcionales o de los algoritmos de estructuras de datos con representaciones y animaciones de la evolución del programa puede aumentar la motivación de los alumnos y aumentar su comprensión de la naturaleza de los programas y algoritmos.

Para que la visualización de algoritmos resulte efectiva desde el punto de vista docente, las representaciones, animaciones, y la propia herramienta de visualización deben cumplir una serie de características, entre ellas la capacidad de interacción por parte del usuario y la calidad y claridad de lo representado, mostrando de la forma más intuitiva posible y acompañando las ejecuciones de texto explicativo.

6 Planteamiento del proyecto

El objetivo de este proyecto es la elaboración de un intérprete de visualizaciones. Tal y como se describe anteriormente en el resumen, este proyecto plantea la implementación de una herramienta que sea no sólo capaz de interpretar un lenguaje funcional, sino de visualizar los estados de ejecución del programa y la ejecución de algoritmos involucrados en el programa interpretado.

El objetivo de dicha herramienta no es meramente académico. Aparte de la investigación de visualización de software, se pretende llegar a obtener una herramienta que pueda ser útil para enseñar programación funcional y los algoritmos sobre estructuras de datos más usados en programación funcional.

6.1 Planteamientos iniciales

El análisis de requisitos de nuestro programa se llevó a cabo a partes iguales entre especificación directa de requisitos por parte del profesor director del proyecto y a través de documentación bibliográfica y del análisis de herramientas similares a la que diseñamos, como es el caso de **Vital** (véase [Vit]) y **WinHIPE**.

La documentación se realizó no sólo para aclarar los objetivos del proyecto, sino que se revisó también la documentación relativa a otros proyectos similares en la Web para tener una idea exacta de los requisitos necesarios para realizar visualizaciones de software adecuadas a la naturaleza de nuestro proyecto (ver la sección *Diseño de visualizaciones de algoritmos* para más información sobre las características de una correcta visualización de software y algoritmos).

En los aspectos relativos a la organización, llegamos al acuerdo entre profesor y alumnos de usar un modelo mixto de modelo de proceso incremental y programación extrema. Debido a lo complejo de la obtención de requisitos para una correcta visualización de software que cumpla con los objetivos propuestos, se optó por un modelo de proceso que pudiese aportar la capacidad de proporcionar de manera rápida un conjunto limitado de funcionalidades, que se refinasen, corrigiesen y expandieran progresivamente hasta obtener un resultado satisfactorio.

El modelo de programación extrema aportaría agilidad a los cambios de requisitos o mejoras de los sucesivos prototipos, mientras que el modelo incremental permitiría realizar tantas versiones como fuesen necesarias de cada prototipo para optimizar su funcionamiento antes de ampliarlo con nuevas capacidades.

Este modelo de trabajo se completaba con reuniones semanales entre profesor y alumnos para planificar los siguientes aspectos a desarrollar, mostrar prototipos, discutir decisiones de diseño y solucionar problemas generales sobre la marcha del proyecto.

Una vez aclarados los aspectos relativos a planificación, los miembros del proyecto conseguimos las infraestructuras software necesarias para gestionar tanto los ficheros fuente del propio proyecto como la documentación asociada.

Usamos servidores de alojamiento de datos gratuitos como CVSDude (http://www.cvsdude.org) para montar un repositorio de datos donde alojar el código fuente de nuestro proyecto. Para la comunicación, usamos listas de correo, y para la gestión de la documentación creamos un grupo de trabajo donde poder almacenar documentación, bibliografía, documentos con actas de todas las reuniones e incluso versiones antiguas de prototipos del proyecto.

6.2 Decisiones de diseño preliminares

La primera decisión de diseño seria fue concretar el lenguaje fuente para el intérprete. En la sección *7 Descripción del lenguaje adoptado para el intérprete*, se explica con más detalle el lenguaje adoptado y su visualización.

En el modelo imperativo la representación de un programa puede hacerse de una forma relativamente fácil, debido a que los programas imperativos almacenan de forma clara y definida el valor de las variables del programa. La evolución del programa se ve marcada por los cambios de los valores de esas variables de un estado al siguiente. Por el contrario, en el modelo funcional no existe esa facilidad para describir el estado de la ejecución de un programa a través de los valores de sus variables. La representación del programa debe venir dada en su lugar por la reescritura del conjunto de programa y datos como una única entidad.

Dado que por este motivo la representación de programas imperativos tiene poco que ofrecer a la investigación sobre representación de programas, decidimos centrarnos en un lenguaje funcional como lenguaje fuente de nuestro intérprete.

Para llegar a este lenguaje fuente consultamos las características de algunos lenguajes declarativos, tales como Hope, Haskell, ML y Clean, para analizar qué características eran deseables tanto para el propio lenguaje fuente como para su visualización.

Los lenguajes Clean y Hope fueron pronto descartados por tener un uso minoritario en el mundo de la programación funcional. Haskell y ML fueron más difíciles de evaluar, ya que la idoneidad de algunas de sus características, como la inferencia de tipos, eran muy complicadas de valorar para nuestro proyecto. En principio Haskell era un lenguaje cuyas características conocíamos mejor que las de ML, pero aún así tenía particularidades, como las mónadas, evaluación perezosa, que no eran deseables para nuestro proyecto.

Valoramos algunas posibilidades, como la opción de crear un lenguaje similar e híbrido de Haskell y ML pero no igual, y que tuviese sólo características que nos interesaran para el lenguaje fuente de nuestro intérprete (sin mónadas, sin evaluación perezosa…). Finalmente, optamos por esta última opción.

Para elegir un lenguaje de implementación nos basamos, como criterio principal de nuestra búsqueda y selección, en encontrar las características que harían más fácil la implementación del intérprete a la par que nos diesen buenas facilidades a la hora de dibujar y representar los estados del intérprete.

En un principio, consideramos Haskell como la alternativa más prometedora para programar el intérprete. Sin embargo Haskell fue descartado, ya que si bien la

programación del intérprete sería sencilla y elegante, este lenguaje presentaría muchos problemas a la hora de dibujar y mostrar la ejecución.

Nuestras siguientes opciones fueron los lenguajes C++ y **Java**. Estos dos lenguajes presentaban el problema opuesto al que presentaba la elección de Haskell como lenguaje de implementación. Estos lenguajes harían más sencilla la parte de mostrar la ejecución y ofrecerían mejores posibilidades en ese punto, sin embargo, al ser lenguajes imperativos, la parte del intérprete sería más complicada.

Dado que el análisis de requisitos del proyecto había dejado clara la importancia de mostrar adecuadamente el programa, decidimos decantarnos por los lenguajes que proporcionasen las mayores posibilidades a la hora de dibujar. Elegir un lenguaje imperativo para implementar el intérprete de un lenguaje funcional, aunque no parecía una decisión ventajosa, no entraba en conflicto con nuestros requisitos.

Para compensar las deficiencias de los lenguajes imperativos a la hora de implementar el intérprete, consideramos el uso de herramientas tipo LEX y YACC, para la generación de analizadores léxicos y sintácticos que simplificasen el trabajo.

Las consideraciones finales inclinaron la balanza a favor del lenguaje Java. Primero estaba la necesidad de portabilidad, para la ejecución en distintas plataformas y entornos Web. Además, teníamos la profunda certeza de que para representar el estado y la evolución de un estado hacia el siguiente, generaríamos gran cantidad de datos, y C++ dificultaría las tareas de recolección de basura necesarias para evitar tener problemas de memoria.

Al decidirnos por Java como lenguaje de implementación, consideramos el uso de la herramienta **JavaCC** para ayudarnos en la construcción del intérprete. JavaCC [Jav] es una herramienta que lee la especificación de una gramática y la convierte a un programa Java que puede reconocer cadenas de la gramática a la vez de hacer un análisis sintáctico de la secuencia leída. JavaCC aportaba su vez otra serie de herramientas como **JJTree**, que permite construcción de árboles.

El uso de esta herramienta contribuiría a reducir los problemas que presentaba la elección de un lenguaje imperativo para la implementación del intérprete de un lenguaje funcional. Al permitir el uso de Java, cuyas características aportaban enormes ventajas para la representación del estado interno del intérprete, decidimos seguir adelante con la elección de usar Java y JavaCC.

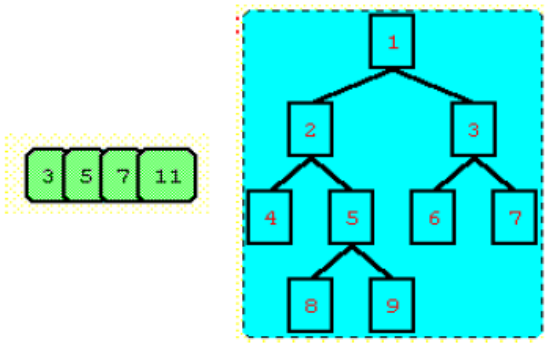
6.3 Decisiones de visualización preliminares

Una buena visualización debería ser gráfica y textual. Por ejemplo **WinHipe** dispone de una representación gráfico-textual, permite parametrizar la visualización y entre otras facilidades reducir expresiones de un tamaño considerable en las partes meramente relevantes de una evaluación, mediante distintas técnicas de ojo de pez.

La manera más simple de visualizar las expresiones es textual, siendo una representación muy común para muchos problemas (por ejemplo matemáticos). Pero la situación cambia cuando queremos mostrar algunas estructuras de datos, por ejemplo un árbol:

```
Node (Node (Empty,
            5,
            Empty),
      3,
      Node(Node(Empty,
                11,
                Empty),
           7,
           Empty))
```

Como vemos, se ha sangrado la estructura para una mejor comprensión, pero aun así la comprensión de estructuras más complejas combinando árboles, aplicaciones de funciones, etc. se hace menos legible. Una solución más expresiva radica en tener representaciones gráficas para las estructuras de datos, y disponer de una mezcla de representación gráfica y textual.

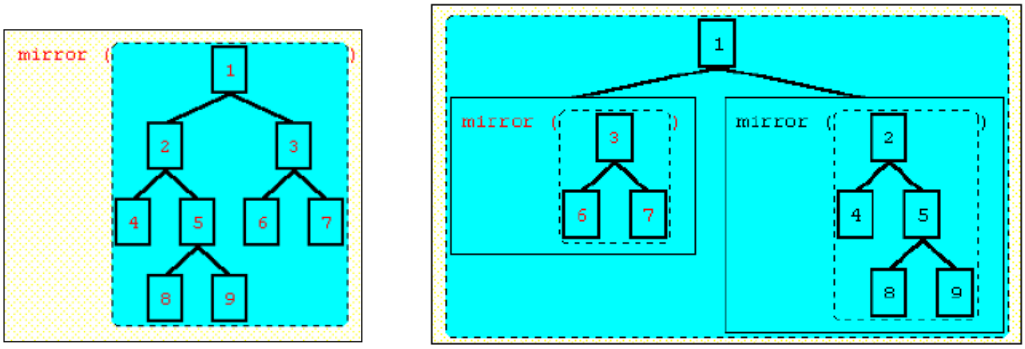


Visualización de una lista y un árbol en WinHipe

Estas estructuras deben cumplir también ciertas características, por ejemplo un árbol puede omitir los árboles vacíos, conviene que el nodo padre esté centrado con respecto a sus hijos, interesa una cierta simetría y aprovechamiento del espacio, etc.

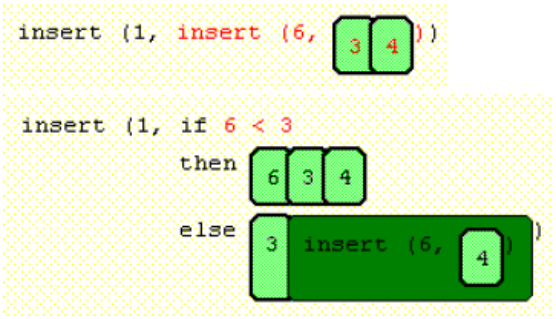
Por otro lado disponer de una mezcla de representación visual y gráfica como **WinHipe**, permite una mayor compresión de estados de evaluación de expresiones más complejas:

```
dec mirror: TREE(alpha) -> TREE(alpha);
--- mirror Empty <= Empty;
--- mirror (Node(lt,x,rt)) <=
       Node(mirror(lt),x,mirror(rt));
```



Visualización en WinHipe

```
dec insert: num#list(num) -> list(num);
--- insert (e,[])    <= [e];
--- insert (e,x::l) <=
      if e < x then e::(x::l)
      else x::insert(e,l);

dec insertsort: list(num) -> list(num);
--- insertsort  []    <= [];
--- insertsort (x::l) <=
       insert(x,insertsort(l));
```



Visualización en WinHipe

También conviene disponer de la capacidad de fijar focos de atención, porque aumentan la expresividad visual. Por ejemplo árboles de gran tamaño, no tienen por qué ser representados completamente; podemos también optar por poner puntos suspensivos en aquellas ramas no relevantes para una mejor comprensión de la visualización.

6.4 Decisiones de visualización tomadas

En **Cardida,** para la visualización hemos utilizado como herramienta GraphViz (GraphViz es un editor de grafos). Dicha herramienta, que puede consultarse en [Gra], es un programa que toma la descripción de un grafo a través de una simple descripción textual directa de la estructura del mismo y es capaz de generar su representación correspondiente en diversos formatos útiles, desde simples imágenes hasta páginas Web.

En concreto, GraphViz posee una aplicación denominada DOT que dibuja representaciones de grafos teniendo en cuenta las características del grafo para evitar el cruce de aristas, minimizar la longitud de las mismas o colocar los nodos del grafo de forma que se ahorre espacio al representar el grafo.

La aplicación, permite no sólo especificar el grafo con relativa facilidad, sino que además permite controlar prácticamente todas las características involucradas en la representación como tamaño de los nodos, tabulación de los mismos, edición de textos en los nodos, tamaño y tipo de letra, colores…

En la siguiente figura puede apreciarse un ejemplo de las capacidades de este programa para la representación de estructuras de datos.

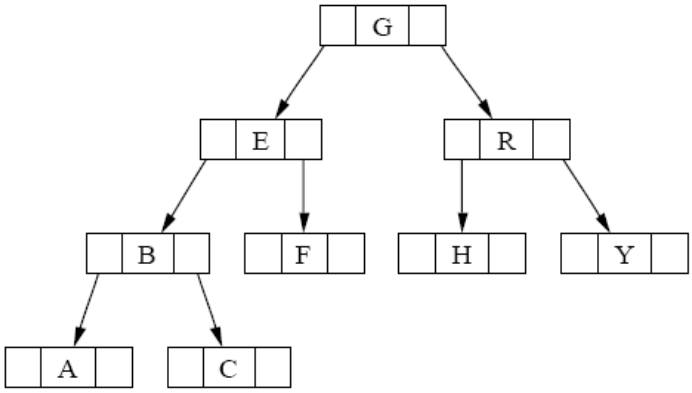


Figura: Representación de un árbol binario con DOT

Los motivos que nos han llevado a apoyarnos en una herramienta que realizase representaciones gráficas en lugar de implementar directamente esa funcionalidad en el proyecto, como inicialmente había sido nuestra intención, han sido la falta de tiempo y las dificultades encontradas durante la implementación del intérprete.

Este proyecto, por su propia naturaleza, está limitado en tiempo y en esfuerzo. La normativa de la asignatura de *Sistemas Informáticos* de la Universidad Complutense de Madrid impone un plazo de entrega no prorrogable del proyecto, así como la restricción de formar grupos de exactamente tres alumnos.

Al haber invertido mucho más esfuerzo del inicialmente planificado para realizar el intérprete, y al vernos limitados en la cantidad de tiempo y esfuerzo que se podía dedicar para obtener un resultado satisfactorio en el plazo previsto, optamos por apoyarnos en la herramienta DOT para agilizar el desarrollo de la aplicación.

El análisis de esta herramienta revelaba que, mediante una especificación cuidadosa de la estructura a dibujar, DOT mantenía la mayor parte de los requisitos que considerábamos esenciales para una representación efectiva del programa interpretado. Por tanto, decidimos incorporar esta herramienta al desarrollo de nuestro proyecto para solucionar los problemas ocasionados durante la implementación del intérprete.

No obstante el uso de esta herramienta no esta exenta de inconvenientes. Ésta es una herramienta preparada para la visualización de grafos y por tanto nos limita a la hora de poder disponer de una capacidad visual como la ofrecida por **WinHipe.** Por ejemplo entre otras carencias que se pueden encontrar, una representación gráfico-textual como hace **WinHipe** se hace casi imposible. Por tanto una mejora esencial de futuras versiones en **Cardida,** sería la implementación de algoritmos de visualización sin ningún tipo de herramienta.

6.5 Análisis

El punto de inicio para el análisis de requisitos de nuestro proyecto era determinar qué aspectos de la interpretación de un programa funcional queríamos mostrar. Estos aspectos determinarían las características que habría de tener nuestro lenguaje fuente.

La primera consideración una vez adoptado un lenguaje fuente, era incluir como tipos de datos básicos, además de enteros y booleanos, los tipos reales, caracteres y cadenas de caracteres. Sin embargo, debido a que en programación funcional los aspectos más interesantes son la manipulación de listas y árboles, decidimos rescribir la gramática del lenguaje fuente para centrarnos sólo en los tipos básicos entero y booleano y en los tipos construidos lista y árbol binario.

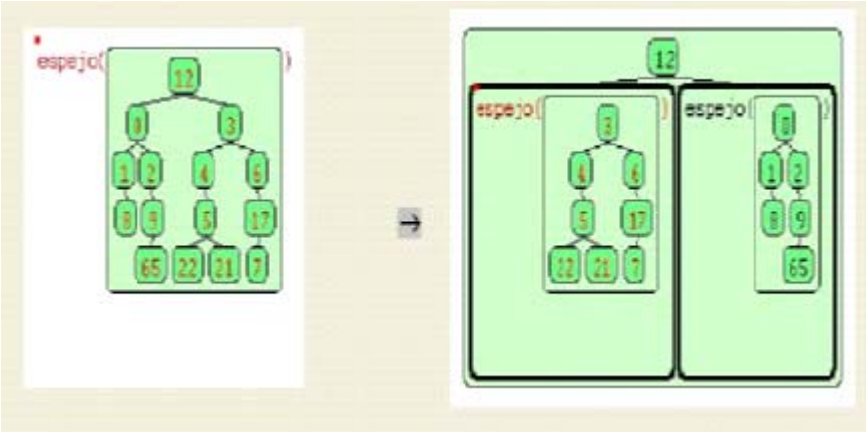
Para precisar los requisitos de la representación tomamos como referente el programa **WinHipe**. La siguiente figura muestra un ejemplo de reescritura del código realizada por WinHipe.

```
insert  (4,  [1, 2, 5, 9]): list (num)

-->

if 4< 1
then [4, 1, 2, 5, 9]
else 1::insert(4, [2, 5, 9])

-->

if false
then [4, 1, 2, 5, 9]
else 1::insert(4, [2, 5, 9])

-->

1::insert (4, [2, 5, 9])

-->

1::(if 4<2
then [4, 2, 5, 9]
else 2::insert(4, [5, 9])

-->

...

-->
```

```
1::2::insert(4, [5, 9])
-->
...
```

Considerando que esta forma de hacer la reescritura del programa que se está interpretando resulta clara para seguir los pasos de interpretación y la evolución del algoritmo, decidimos realizar los pasos de reescritura para nuestro intérprete de una forma similar a la de WinHipe.

Análogamente, decidimos optar por un estilo de representación similar al de WinHipe, que cumple la mayor parte de los requisitos necesarios para una representación efectiva de un programa.



Ejemplo de visualización de un programa con WinHipe

Aparte de la forma de realizar la reescritura del código fuente y la representación del mismo se fijaron otras restricciones. Además de mostrar una ejecución paso a paso del programa interpretado, se consideró de interés que la evolución de las imágenes mostradas pudiese ser tanto una ejecución paso a paso manual por parte del usuario como una ejecución automática. Esta ejecución automática debería avanzar a una velocidad que permitiese observar los cambios producidos por el algoritmo a la vez que este avanzaba.

Para facilitar la comprensión del algoritmo, se optó por hacer que la ejecución de la secuencia de imágenes mostrase un paso y el siguiente, lo que permite comparar ambos estados.

Con respecto a las páginas html que generara nuestra aplicación, deberían tener secciones de descripción del problema, descripción del algoritmo y texto del programa, además de la propia animación.

Respecto a la diversidad funcional que podría tener una aplicación de este estilo, nos centramos simplemente en:

- Generar html: el usuario tendría la posibilidad de crear una página html con las secciones de descripción del problema, descripción del algoritmo, texto del programa y título de la página. La animación generada para la página tendría la posibilidad de avanzar automáticamente o paso a paso, además de visualizar el estado de una evaluación anterior y posterior, con la finalidad de poder observar los cambios producidos.

- Editar, abrir y guardar un nuevo programa.

- Reescribir un programa: una vez editado o abierto un programa, tener la posibilidad de realizar y posteriormente ver la reescritura completa del programa.

- Visualizar un programa: una vez editado o abierto un programa, se podrá realizar la visualización del programa. Una vez realizada la misma, el usuario dispondría de la posibilidad de navegar por los sucesivos estados de la evaluación.

- Cambiar propiedades de la visualización: el usuario antes de realizar la visualización de un programa, tendría la posibilidad de cambiar la apariencia de la misma, cambiando los colores de los datos y aplicación de funciones si así lo deseara.

7 Descripción del lenguaje adoptado para el intérprete

Tal como se indicaba en la sección 6.2 *Decisiones de diseño preliminares*, el lenguaje fuente escogido para nuestro intérprete de visualización es un lenguaje subconjunto de Haskell con algunas características propias introducidas y similares a las de WinHipe. A continuación pasaremos a explicar algunas de sus características.

7.1 Descripción informal del lenguaje fuente

Antes de definir sus aspectos léxicos y sintácticos, creemos aconsejable mostrar un par de programas de ejemplo para dar una idea general de cómo es un programa escrito en el lenguaje fuente del intérprete.

A continuación se presenta un programa simple para el cálculo del factorial de un número entero (Programa de ejemplo 1)

```
module Factorial where

    decfun factorial :: Int -> Int ;

    fun factorial n

                | 0 = 1

                | n = n * fun factorial (n - 1)

    end_fun

    main :: Int ;

    main = fun factorial 3

end
```

Programa de ejemplo 1

Como en cualquier lenguaje de tipo funcional, el programa se compone únicamente de declaraciones de funciones. Las funciones se definen por medio de ecuaciones, declarando en la cabecera de la función los tipos de sus parámetros y su resultado.

En el ejemplo vemos cómo se declara una función *factorial* que tiene un argumento de tipo entero y devuelve un valor de tipo entero. Cada ecuación de la función *factorial* tiene asociada una condición, que discrimina el valor del argumento de la función. Al igual que en Haskell, las condiciones son evaluadas en el orden en el que están escritas. La función principal *main* sólo contiene una llamada a la función *factorial* para calcular el valor del factorial de 3.

Pasemos a un ejemplo no tan básico. El segundo programa de ejemplo que presentamos (Programa de ejemplo 2) calcula el recorrido en preorden de un árbol.

```
module PreOrden where

 decfun concat :: List(alpha) -> List(alpha) -> List(alpha);

    fun concat xs ys

        | [] ys  = ys

        | ConsList(x, restXs) ys = ConsList(x,fun concat restXs ys)

 end_fun


 decfun preorden :: Tree(alpha) -> List(alpha) ;

 fun preorden tree

     | Empty = []

     | Node(left, x, right) = fun concat [x] (fun concat

                                                  (fun preorden left)

                                                  (fun preorden right)

                                              )

 end_fun


 main :: List(alpha);

 main = fun preorden Node(Node(Empty,3,Empty),2, Node(Empty,2,Empty))

end
```

Programa de ejemplo 2

En este programa se han declarado dos funciones. La primera es la función *concat,* una función auxiliar que concatena dos listas. La función *preorden* recibe un árbol y devuelve una lista con el preorden de los valores del árbol.

La palabra reservada *Empty* designa el árbol vacío. Para la distinción de casos de la función *concat* podemos observar que la primera ecuación comprueba si la primera lista es vacía, y en caso de que lo sea devuelve la segunda lista. En la segunda ecuación se puede observar el constructor de las listas en nuestro programa.

La función *preorden* recibe un árbol y devuelve una lista. Si el árbol es vacío, se devuelve la lista vacía. En caso contrario se devuelve la concatenación de la lista formada por la raíz, el preorden del subárbol izquierdo y el preorden del subárbol derecho. En este

ejemplo se pueden observar detalles del lenguaje fuente, como la declaración de una lista unitaria ([x]) y de un árbol vacío.

De igual forma que el anterior programa, la función principal contiene un pequeño ejemplo. El árbol que se le pasa por parámetro es un árbol cuya raíz es el número 2 y sus subárboles izquierdo y derecho son respectivamente los valores 3 y 7.

Podemos apreciar entonces que, en el lenguaje fuente del intérprete, existe siempre una cabecera de programa con el nombre del mismo (*module* NombrePrograma *where*…). Después viene una sección de declaración de funciones. La cabecera de las funciones se indica con la palabra reservada *decfun*, la propia función con *fun* y el final de dicha declaración con *end_fun*. Tras la sección de declaraciones viene la función principal, indicada con *main*. Además de las distinciones de casos presentadas en los dos programas de ejemplo, dentro del cuerpo de una función pueden aparecer cláusulas *let*, *where*, instrucciones condicionales *if-then-else* …

7.2 Aspectos léxicos

A continuación se presentan los aspectos léxicos del lenguaje fuente.

Símbolos permitidos y signos de puntuación

```
ARROW =          \->
LBRACE =         {
RBRACE =         }
LPAREN =         \(
RPAREN =         \)
COMMA =           ,
TWODOT =         :
SEMICOLON =      ;
TWODOUBLEDOT =   ::
VERTICAL_BAR =   \|
RBRACKET =       ]
LBRACKET =       [
```

Palabras reservadas

```
BOOL =       Bool
DECFUN =     decfun
ENDFUN =     end_fun
ELSE =       else
END =        end
FALSE =      FALSE
FUN =        fun
IF =         if
INT =        Int
MAIN =       main
MODULE =     module
THEN =       then
TRUE =       TRUE
WHERE =      where
NODE =       Node
```

```
EMPTY =         Empty
LET=            let
IN =            in
LIST =          List
CONS_LIST =     ConsList
TREE =          Tree
```

Operadores

```
AND =           and
ASSIGN =        =
DIV =           div
GT =            >
LT =            <
EQ =            ==
LE =            <=
GE =            >=
NE =            <>
PLUS =          \+
MINUS =         \-
NOT=            not
OR =            or
MOD =           mod
MUL =           *
```

Identificadores y números

```
INT_LITERAL = (-? [1 - 9]  ([0 - 9])* ) | 0

DIGIT = ['0' – '9']

VARID = ['a' – 'z']  (['a' – 'z'] | DIGIT | ['A' – 'Z'] | '_')* (' ')?
CONID = ['A' – 'Z'] (['a' – 'z'] | DIGIT | ['A' – 'Z'] | '_')*
```

VARID son los identificadores para las variables y funciones. En nuestro lenguaje fuente sufren la restricción de empezar con minúscula. CONID son los identificadores de las constructoras de los tipos de datos. Para distinguirse de variables y de funciones, deben empezar por mayúscula.

7.3 Aspectos sintácticos: Gramática del lenguaje

En este apartado expresamos la sintaxis del lenguaje fuente a través de las producciones de una gramática independiente del contexto.

```
Module::=  module CONID where Body end

Body::=  TopDecls FunMain

TopDecl::=  TopDecl TopDecls

            | λ
```

TopDecl::= Decl

Decl::= FunDecl FunBody

FunDecl::= **decfun** VARID **::** Type **;**

FunBody::= **fun** FunLhs Rhs **end_fun**

AtypeSequence::= Atype AtypeSequence

| λ

TupleType::= **(** TypeList **)**

TypeList::= Type

| Type **,** TypeList

Type::= Btype

| Btype **->** Type

Btype::= Atype

| Atype RestBtype

RestBtype::= Atype

| Atype RestBtype

Atype::= TupleType

| VARID

| Bool

| Int

| List **(** Type **)**

| Tree **(** Type **)**

FunLhs::= VARID PatSequence

PatSequence::= Pat PatSequence

| λ

Pat::= Literal

| ListPat

| TreePat

ListPat::= **[]**

| **ConsList(** Pat **,** Pat **)**

TreePat::= **Empty**

| **Node (** Pat **,** Pat **,** Pat **)**

Rhs::= **=** Stament

| CaseBody

CaseBody::= **|** PatSequence **=**

| **|** PatSequence **=** CaseBody

FunMain::= FunMainDecl FunMainBody

FunMainDecl::= **main ::** Type **;**

FunMainBody::= **main =** Statement

Stament::= **let {** Defs **} in {** Exp **}**

| Exp

| Exp **where {** Defs **}**

| **if** Exp **then** Statement **else** Statement

Defs::= VARID **=** Statement **;**

| VARID **=** Statement **;** Defs

Exp::= SimpleExp

| SimpleExp RelOp SimpleExp

SimpleExp::= Term

| Term RestSimpleExp

RestSimpleExp::= Term

| Term RestSimpleExp

RestSimleExp::= AddOp Term

| AddOp Term RestSimpleExp

Term::= Factor

| Factor RestTerm

RestTerm::= MulOp Factor

| MulOp Factor RestTerm

Factor::= VARID

| **fun** VARID Factor

| TupleExp

| Literal

| ListFactor

| TreeFactor

| **not** Factor

ListFactor::= **[]**

| **[** Exp, Exp…. **]**

| **ListCons(**Exp**,**Exp**)**

TreeFactor::= **Node (** Exp **,** Exp **,** Exp **)**

| **Empty**

Literal::= INT_LITERAL

| **TRUE**

| **FALSE**

TupleExp::= **(** ExpList **)**

ExpList::= Exp

| Exp **,** ExpList

RelOp::= **= =**

|**< =**

|**> =**

|<>

| **>**

| **<**

AddOp::= **+**

| -

|**or**

MulOp::= **and**

| **mod**

| **div**

| *

8 El intérprete

A modo de introducción diremos que un *intérprete puro* es un programa que analiza y ejecuta simultáneamente un programa escrito en un determinado lenguaje, llamado lenguaje fuente.

Un intérprete en general tiene dos entradas: un programa P, escrito en un lenguaje fuente, que denotaremos LF, y los datos de entrada. A partir de dichas entradas, mediante un proceso de *interpretación*, se van produciendo unos resultados.

En la Figura 1 se presenta el esquema general de un intérprete visto como una caja negra.

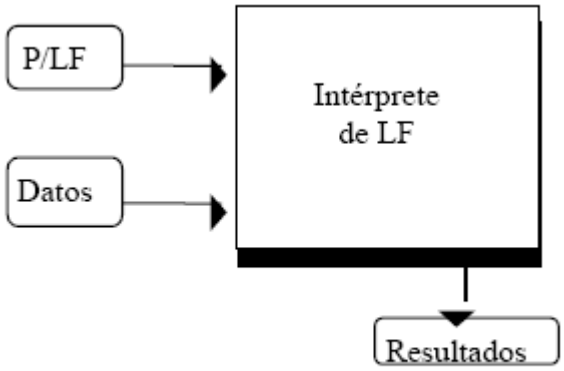


Figura 1: Esquema general de un intérprete

Los compiladores, a diferencia de los intérpretes, transforman el programa a un programa equivalente en un código objeto (fase de compilación), y en un segundo paso generan los resultados a partir de los datos de entrada (fase de ejecución).

En nuestro caso, los datos son reemplazados por una expresión inicial (main) y los resultados se corresponden con un proceso de reescritura. Hay que destacar que en modelo imperativo los resultados producidos quedan definidos fácilmente por los cambios de valores de las variables. Por el contrario, en el modelo funcional no existe esa facilidad a la hora de describir un resultado o estado de ejecución. La razón es que los distintos resultados vienen expresados en el proceso de reescritura y por tanto el conjunto de programa y datos son tratados como una única entidad.

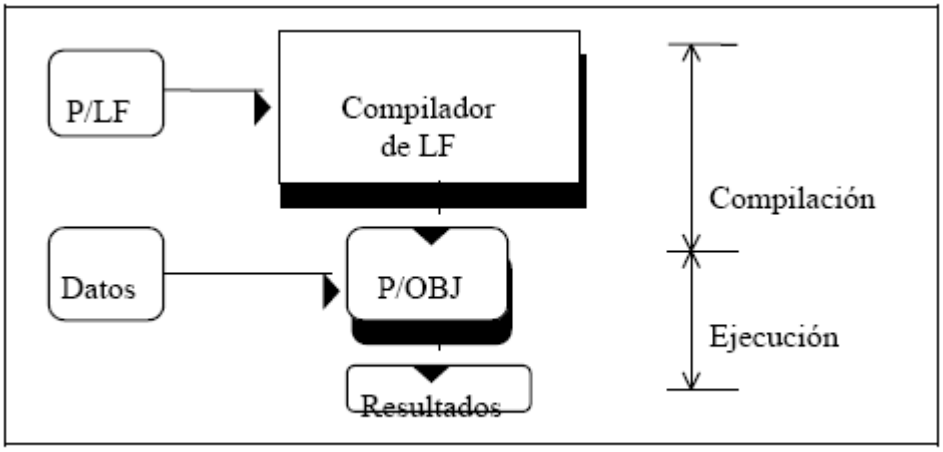


Figura 2: Esquema general de un compilador

Nuestro objetivo en el proyecto ha sido desarrollar un intérprete visual de un lenguaje funcional. Para ello, primero hemos tenido que concretar una representación abstracta del programa que se quiere interpretar. Esta representación, que pasaremos a comentar posteriormente en el apartado 8.2 *Técnicas de interpretación*, nos ha ayudado tanto en la evaluación e interpretación como en la visualización del programa.

8.1 Estructura de un intérprete

A la hora de construir un intérprete es conveniente utilizar una Representación Interna (RI) del lenguaje fuente a analizar. De esta forma, la organización interna de la mayoría de los intérpretes se descompone en los siguientes módulos:

- *Traductor a Representación Interna*: Toma como entrada el código del programa *P* en Lenguaje Fuente, lo analiza y lo transforma a la representación interna correspondiente a dicho programa *P*.

- *Representación Interna (P/RI)*: La representación interna debe ser consistente con el programa original. Entre los tipos de representación interna, los árboles sintácticos son los más utilizados y, si las características del lenguaje lo permiten, pueden utilizarse estructuras de pila para una mayor eficiencia.

- *Tabla de símbolos*: Durante el proceso de traducción, es conveniente ir creando una tabla con información relativa a los símbolos que aparecen. La información a almacenar en dicha tabla de símbolos depende de la complejidad del lenguaje fuente. Se pueden almacenar etiquetas para instrucciones de salto, información sobre identificadores (nombre, tipo, línea en la que aparecen, etc.) o cualquier otro tipo de información que se necesite en la etapa de evaluación.

- *Evaluador de Representación Interna*: A partir de la Representación Interna anterior y de los datos de entrada, se llevan a cabo las acciones indicadas para obtener los resultados. Durante el proceso de evaluación es necesario contemplar la aparición de errores.

- *Tratamiento de errores*: Durante el proceso de evaluación pueden aparecer diversos errores como desbordamiento de la pila, divisiones por cero, etc. que el intérprete debe contemplar.

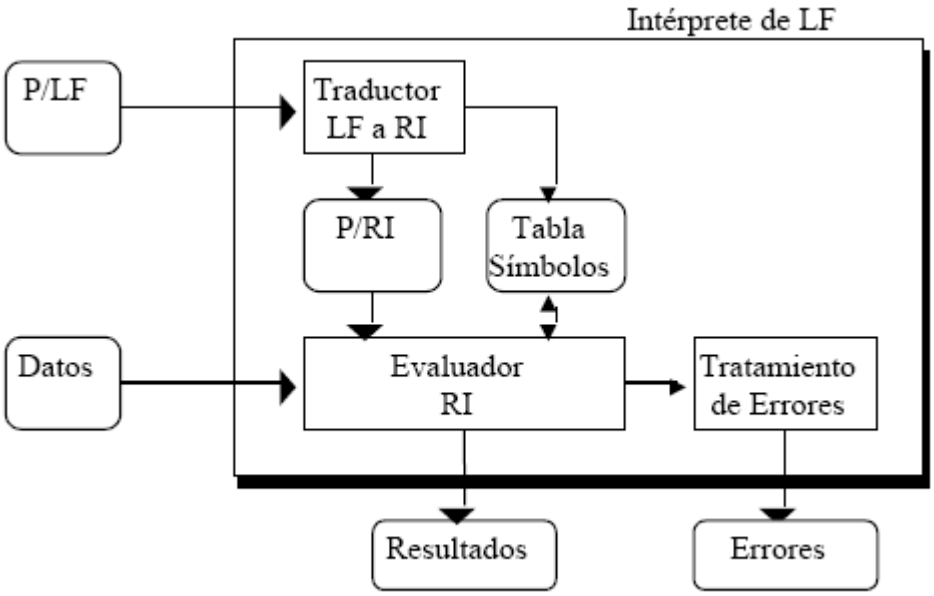


Figura 3: Organización interna de un intérprete

En nuestro caso carecemos de una tabla de símbolos en sí, pero disponemos de un **entorno**, que viene a sustituir la tabla de símbolos y cuya utilidad comentaremos en el apartado de *Técnicas de interpretación*.

Respecto al tratamiento de errores, podrían presentarse errores de tipo léxico, aritméticos, etc. de los cuales informamos al usuario para su posible corrección.

Como traductor utilizamos *JavaCC*, con el cual conseguimos el árbol sintáctico representado por el programa, y el cual transformaremos en otro árbol sintáctico abstracto para la posterior evaluación-interpretación y visualización del programa, siendo este último árbol de sintaxis abstracta nuestra representación interna definitiva (RI). Por último nuestro *evaluador RI* será el encargado de realizar la reescritura y visualización, basado en las técnicas de interpretación comentadas a continuación.

8.2 Técnicas de interpretación

Como hemos ido viendo en el apartado anterior, en este punto trataremos de fijar el código intermedio (RI), así como la técnica de interpretación llevada a cabo por el evaluador *RI*, que viene a ser lógica de negocio del intérprete. Para una mejor comprensión, como referencia en el apéndice se adjunta un pequeño intérprete desarrollado en Haskell, y que ha sido nuestra guía a la hora de la implementación en java de nuestro intérprete.

A continuación iremos definiendo el intérprete, y en el apartado correspondiente a implementación y diseño comentaremos y explicaremos como se ha desarrollado una

versión definitiva de éste, así como el porqué de las decisiones tomadas. En este apartado simplemente nos centraremos en los fundamentos teóricos.

Código intermedio

El código intermedio del programa podemos representarlo fácilmente en Haskell, Hope…. Éste lo podemos generar mediante un parser o directamente transformando el árbol sintáctico de un programa fuente. Hay que tener en cuenta que el código intermedio, dependiendo del interprete utilizado, es decir si es perezoso o impaciente son distintos. Nuestro intérprete es impaciente. Además el código intermedio nos proporciona una especificación semántica del programa.

Un ejemplo del código intermedio para las expresiones de nuestro lenguaje representadas en Haskell es el siguiente:

```
data Expresion = Var    Ident
               | Entero Int
               | Logico Bool
               | Lambda [Ident] Expresion
               | Aplic  Expresion [Expresion]
               | Primit Ident
               | Where  Expresion [(Ident, Expresion)]
               | Condic Expresion Expresion Expresion
               | Clausura Expresion Entorno
              deriving Show
```

Luego expresiones en nuestro código intermedio serían, por ejemplo, de la siguiente forma:

5 + 7 sería: *(Aplic (Primit "+") [Entero 5, Entero 7])*

If (3 > 2) then 3 else 5 sería:

(Condic (Aplic (Primit ">") [Entero 3, Entero 2]) (Entero 3) (Entero 5))

Este tipo de expresiones pueden concebirse como una estructura de árbol abstracto, donde por ejemplo "5 + 7" puede verse como un árbol con un nodo "Aplic" como raiz y nodos "Primit +", "Entero 5", "Entero 7" como hijos.

Un intérprete impaciente

En un intérprete impaciente el paso de parámetros es por valor (call by value). Éste es un método utilizado por la mayoría de los lenguajes imperativos, y consiste en evaluar los

argumentos antes de ejecutar o evaluar el cuerpo de una función con independencia de la necesidad de dicha evaluación. También se conoce como evaluación ansiosa (eager).

Ejemplo:

(λx. x*x)(2+3) → (λx. x*x)5 → 5*5 →25

La mayoría de los intérpretes de lenguajes funcionales, a la hora de realizar la interpretación usan dos funciones llamadas Eval y Apply. Eval se encarga de evaluar las expresiones y Apply de la aplicación de funciones. El intérprete impaciente llevado a cabo implementa la aplicación de funciones dejando el cuerpo de la función intacto y guardando las asignaciones de las variables en una estructura especial llamada entorno, es aquí donde se realiza el paso de parámetros por valor. Esto realiza el mismo efecto que la sustitución de variables.

En el entorno, se va a ir guardando el nombre de la variable así como su valor asignado o expresión en caso de ser una definición de función definida por el usuario.

Luego el entorno va a ser del tipo siguiente: dado el identificador de una función o una variable, nos va a devolver su expresión asociada:

Env :: Id → Exp

Respecto a las funciones Eval y Apply sería el siguiente:

Eval :: Exp → Env → Exp

Apply :: Exp → Env → [Exp] → Exp

Debemos tener en cuenta que la aplicación de funciones va a ser saturada y vamos a saber siempre cuál va a ser el número de expresiones al que vamos aplicar la función.

A continuación iremos definiendo las reglas mediante las cuales se va a ir realizando el proceso de interpretación (las reglas originarias las podremos encontrar en [Fie88]). Estas reglas podrían entenderse como un recorrido a través del árbol abstracto que forman las expresiones representadas en nuestro código intermedio.

8.3 **Reglas para Eval**

1. Aplicación de funciones:

Para la aplicación de funciones, como vemos en la siguiente regla, evaluamos previamente cada expresión y la pasamos como parámetro a la regla *apply* antes de realizar la aplicación de funciones:

```
eval  e@(Aplic fun exprs) env  = apply (eval fun env) [eval e env | e <- exprs]
```

2. Enteros y Booleanos:

La evaluación de un tipo de expresión primitivo será la propia expresión, ya que el ser primitivo (esta en forma normal) no se puede seguir evaluando. Por lo tanto tampoco haremos uso del entorno, de ahí el uso del comodín "_".

```
eval  e@(Entero z) _    = e

eval  e@(Logico p) _   = e
```

Se aplicaría la misma idea para todo tipo primitivo.

3. Identificadores:

En el caso de las variables, evaluaremos la expresión asociada a su identificador, buscando previamente en el entorno dicha expresión ya sea el cuerpo de la función o el valor de una variable:

```
eval  e@(Var   id) env = case lookup id env of

                              Just expr -> eval expr env

                              Nothing   -> error (id ++ " no definido")
```

4. Funciones primitivas:

Caso similar a enteros y boleanos, lo que hacemos es devolver la expresión, ya que es un operador primitivo:

```
eval  e@(Primit id) _  = e
```

5. Condiciones:

Cuando nos encontremos un código intermedio de una condición, previamente evaluaremos la condición, cuyo resultado utilizaremos para evaluar una expresión u otra:

```
eval  e@(Condic cond expr1 expr2) env  = case eval cond env of

                                            Logico True  -> eval expr1 env

                                            Logico False -> eval expr2 env
```

6. Abstracciones Lambda:

Esta regla quizás sea la menos intuitiva, ya que las abstracciones lambda ya están en su forma normal, la razón es que el cuerpo de una expresión lambda puede tener variables libres. Por esta razón no debemos devolver simplemente la expresión, si no que debemos

devolver la expresión junto con su entorno, para poder ligar el valor de las variables libres disponibles en dicho entorno. Esta estructura recibe el nombre de clausura, porque representa una expresión cerrada, es decir sin variables libres.

```
eval e@(Lambda parsFormales expr) env = Clausura e env
```

7. Clausura:

Las clausuras representan funciones que siguen estando en su forma normal. Inicialmente no existen, es decir no forman parte del código intermedio, pero son generadas por el intérprete como resultado de la evaluación de una abstracción lambda. Por lo tanto el resultado de la su evaluación es ella misma.

```
eval  e@(Clausura cl env) env'  = e
```

8. Let y Where:

Las reglas para *where* y *let* simplemente deben evaluar la expresión correspondiente en su entorno adecuado, para lo cual insertamos las nuevas definiciones y realizamos la evaluación bajo el nuevo entorno.

```
eval  e@(Where expr defs) env  = eval expr (defs ++ env)

eval  e@(Let expr defs) env  = eval expr (defs ++ env)
```

8.4 Reglas para Apply

1. *Para funciones predefinidas:*

Para la aplicación de funciones predefinidas simplemente debemos aplicar los argumentos (expresiones) a la función correspondiente. Ejemplo de reglas:

```
apply (Primit op) argumentos = funDe op argumentos

        funDe "+"  [Entero x, Entero y]  = Entero (x+y)

        funDe "+"  [_      , _       ]  = error $ "'+' aplicada a dos pars. NO enteros"

        funDe "+"  _                    = error $ "'+' con inexacto núm. de argumentos"

        funDe "-"  [Entero x, Entero y]  = Entero (x-y)

        funDe "-"  [_      , _       ]  = error $ "'-' aplicada a dos pars. NO enteros"

        funDe "-"  _                    = error $ "'-' con inexacto núm. de argumentos"

        funDe "*"  [Entero x, Entero y]  = Entero (x*y)

        funDe "*"  [_      , _       ]  = error $ "'*' aplicada a dos pars. NO enteros"
```

```
funDe "*"  _              = error $ "'*' con inexacto núm. de argumentos"
```

2. Para clausuras:

La aplicación de funciones, donde la clausura nos da la ligadura a las variables libres, a través del entorno, se realiza copiando en el entorno los parámetros pasados a la regla Apply (de esta manera se consigue el paso de parámetros por valor comentado anteriormente), evaluando la expresión de la lambda abstracción en el nuevo entorno.

```
apply(Clausura(Lambda ids e)env) es = eval e $(zip ids es)++ env
```

Además llegados a este punto, si las funciones son recursivas, para mantener un entorno libre de modificaciones al retorno de las llamadas recursivas, surge la necesidad de utilizar una pila de entornos.

```
apply(Clausura(Lambda ids e)env) es = apilar(env);
                                      eval e $(zip ids es)++ env;
                                      env := desapilar(env);
```

9 Descripción técnica del proyecto

En este apartado se pretende dar una descripción de la arquitectura que sustenta el intérprete de visualizaciones *Cardida*.

A lo largo de todo el desarrollo del proyecto siempre hemos tratado de tomar soluciones mediante patrones de diseño con la finalidad de que los cambios o factores adversos de otra índole que pudieran surgir en el desarrollo del proyecto, así como modificaciones o nuevas funcionalidades a incluir en futuras versiones tuvieran una fácil expansión, mantenimiento y modificación.

Por tanto, a continuación presentamos los principios teóricos y su aplicación al proyecto, así como las soluciones resultantes a las que ha dado lugar la implementación del intérprete. También en el cd adjuntado al presente documento puede encontrarse el código llevado acabo en *Cardida* para una mayor comprensión de algunos aspectos comentados o no en esta sección.

9.1 Breve introducción al uso de patrones de diseño

Los patrones de software son soluciones reutilizables a los problemas que ocurren durante el desarrollo de un sistema de software o aplicación.

La principal ventaja del uso de patrones es que éstos proporcionan un proceso consistente o diseño que uno o más desarrolladores pueden utilizar para alcanzar sus objetivos. También proporciona una arquitectura uniforme que permite una fácil expansión, mantenimiento y modificación de una aplicación.

Existen diferentes tipos de patrones. Dependiendo del nivel conceptual de desarrollo donde se apliquen, se distinguen (de más abstractos a más concretos): patrones de análisis, patrones arquitectónicos, patrones de diseño y patrones de implementación o *idioms*.

Dependiendo del propósito funcional del patrón, se distinguen los siguientes tipos:

- Fundamental: construye bloques de otros patrones.
- Presentación: Estandariza la visualización de datos.
- De creación: Creación condicional de objetos.
- Integración: Comunicación con aplicaciones y sistemas y recursos externos.
- De particionamiento: Organización y separación de la lógica compleja, conceptos y actores en múltiples clases.
- Estructural: Separa presentación, estructuras de datos, lógica de negocio y procesamiento de eventos en bloques funcionales.
- De comportamiento: Coordina/organiza el estado de los objetos.
- De concurrencia: Maneja el acceso concurrente de recursos.

Lejos de querer extendernos sobre los aspectos teóricos de los patrones de diseño, y mencionadas ya las ventajas y motivos que nos han llevado a su uso, recomendamos

[Gam95] como bibliografía complementaria sobre los aspectos relacionados con el uso de patrones.

A continuación, en los sucesivos apartados, iremos explicando cuestiones de diseño e implementación desarrolladas en nuestra aplicación.

9.2 Patrón Modelo-Vista-Controlador

La arquitectura estructural que posee el sistema *Cardida,* está sustentada por el patrón Modelo-Vista-Controlador (MVC). Este patrón es un ejemplo de patrón **arquitectónico estructural**.

La arquitectura del patrón Modelo-Vista-Controlador es un paradigma de programación bien conocido para el desarrollo de aplicaciones con interfaz gráfica (GUI). El principal objetivo de la arquitectura MVC es aislar tanto los datos de la aplicación como el estado (modelo) de la misma, del mecanismo utilizado para representar (vista) dicho estado, así como para modularizar esta vista y modelar la transición entre estados del modelo (controlador). Las aplicaciones MVC se dividen en tres grandes áreas funcionales:

- **Modelo**: la lógica del negocio o servicio y los datos asociados con la aplicación.
- **Vista**: la presentación de los datos.
- **Controlador**: el que atenderá las peticiones para la toma de decisiones de la aplicación.

El propósito del MVC es aislar los cambios. Es una arquitectura preparada para los cambios, que desacopla datos y lógica de negocio de la lógica de presentación, permitiendo la actualización y desarrollo independiente de cada uno de los citados componentes.

A continuación mostramos un esquema de este modelo:

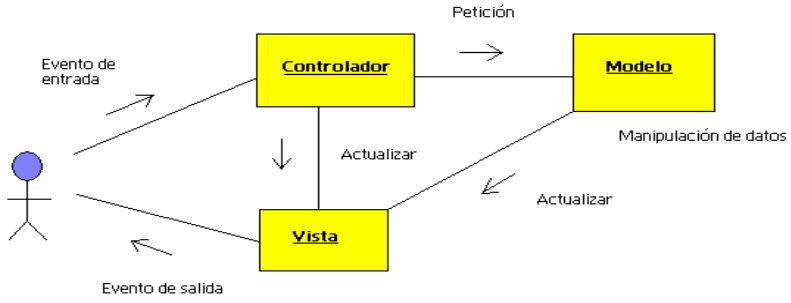


Figura 8.1: Esquema del patrón modelo-vista-controlador

Consultando el código de *Cardida* podemos ver la distribución de paquetes seguida y la correspondencia con la arquitectura propuesta (Figura 8.2).

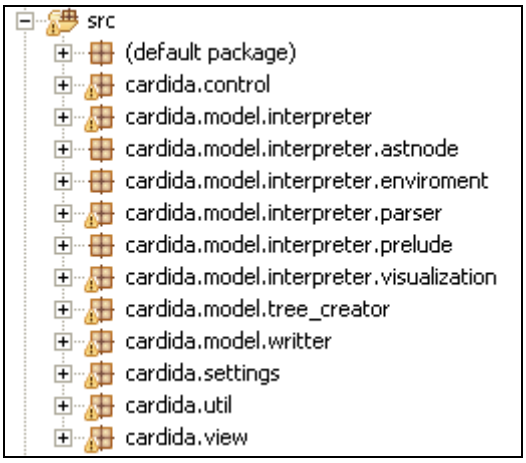


Figura 8.2: Distribución en paquetes del código

Podemos ver que la lógica de negocio se encuentra en todos los paquetes *"cardida.model.*"*. Dentro de la lógica de negocio podemos observar las distintas partes que la componen, bien diferenciadas:

- El intérprete (*"cardida.model.interpreter"*), el evaluador RI.
- El árbol abstracto (código intermedio RI) que se visualiza y es interpretado (*"cardida.model.interpreter.astnode"*).
- El entorno donde se encontrarán las declaraciones de la variables y funciones del usuario (*"cardida.model.interpreter.environment"*).
- El preludio, donde nos encontraremos todas las funciones predefinidas ("*cardida.model.interpreter.prelude"*).
- La lógica encargada de la visualización, reescritura e interpretación de nuestro árbol abstracto (*"cardida.model.interpreter.visualization"*).
- La lógica del parser de la gramática *Cardida* ("*cardida.model.interpreter.parser"*).
- El árbol sintáctico de la gramática y la lógica encargada de su transformación en el árbol abstracto adecuado para la interpretación y visualización ("*cardida.model.tree_creator*").
- Y la lógica encargada de generar los archivos *"html"* ("*cardida.model.writter"*).

La parte correspondiente a la vista y el controlador pueden verse en ("*cardida.view" y "cardida.control"*), aquí podremos encontrar las distintas vistas de usuario disponibles en *Cardida*, así como la gestión de los distintos eventos lanzados por las decisiones tomadas del usuario.

Por otro lado nos encontramos con el paquete *"cardida.settings",* donde encontraremos las clases encargadas de la configuración de *Cardida*, y *"cardida.util"* donde podremos encontrar clases con métodos de uso común en toda la aplicación.

La conexión entre la vista, el controlador y el modelo, se ha realizado mediante interfaces que nos permiten, garantizan y ofrecen un mayor aislamiento entre estos tres módulos funcionales.

Dentro del modelo también hemos adoptado medidas de diseño importantes y que comentaremos a continuación.

9.3 Implementación del código intermedio

La implementación llevada para el código intermedio se ha llevado a cabo mediante una estructura de árbol. Previamente para conseguir este árbol abstracto que representa el código intermedio de nuestro intérprete ha sido necesario la traducción (transformación) del árbol sintáctico formado por cadenas de nuestro lenguaje, y que vienen definidas por las producciones sintácticas expuestas en la sección "7.3 Aspectos sintácticos: Gramática del lenguaje".

Una vez conseguida la transformación, el árbol abstracto se corresponde con el árbol de una expresión de nuestro código intermedio. Este árbol será interpretado mediante las técnicas de interpretación expuestas con anterioridad en las secciones 8.3 y 8.4 para las reglas *Eval* y *Apply* respectivamente, que pueden definirse como recorrido de un árbol, como veremos con posterioridad.

9.4 Creación de operaciones

En la creación de operaciones predefinidas en *Cardida,* en el proceso de interpretación para su ejecución (correspondiente a las reglas de la sección 8.4**,** *Reglas para Apply*) hemos utilizado el patrón de creación ***Abstract Factory***.

El objetivo de este patrón ha sido el de independizar cómo se crean las distintas operaciones predefinidas de *Cardida*, proporcionando una gran flexibilidad acerca de qué se crea (operaciones), quién lo crea, cómo se crea y cuándo se crea. Este patrón es capaz de cambiar una familia de productos por otra y promueve la consistencia entre los productos.

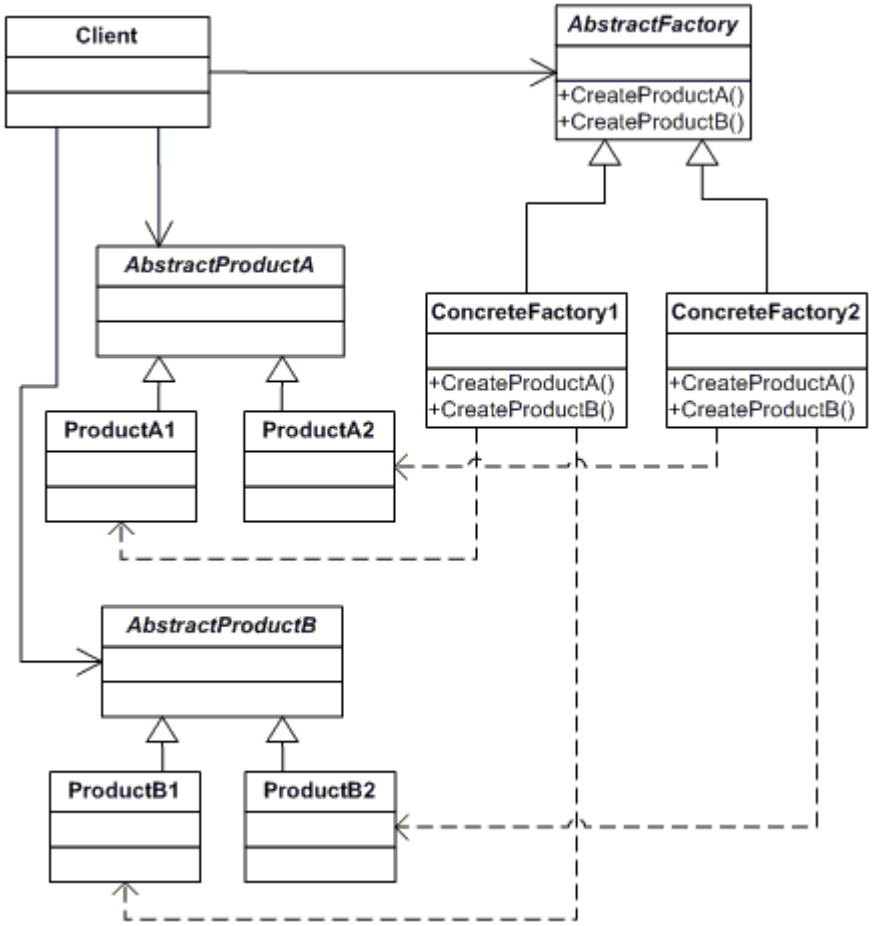


Figura 8.3: Patrón Factory

En la figura 8.3 se puede observar el diagrama de clases que representa el esquema básico del patrón *Factory*. En la figura, *AbstractFactory* define la interfaz para a creación de los productos (en nuestro caso las operaciones) y *ConcreteFactory* implementa dicha interfaz. AbstractProduct declara la interfaz de un producto (las operaciones) y que será creado por *ConcreteFactory*. De esta manera los clientes sólo dependen de las interfaces *AbstractFactory* y *AbstractProduct*.

Además para el comportamiento de cada operación hemos utilizado un patrón de comportamiento ***Strategy.*** De esta manera hemos encapsulado las operaciones de *Cardida* pudiéndolas utilizar en distintos contextos.

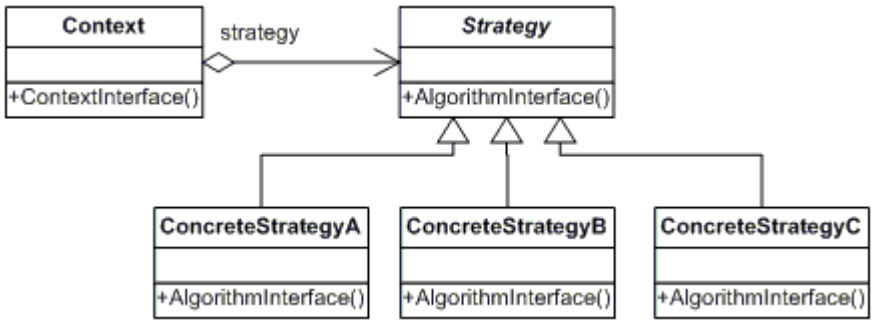


Figura 8.4: Patrón Strategy

Strategy define la interfaz común implementada en cada *ConcreteStrategy*. *Context* es la clase cliente que utiliza las distintas estrategias.

9.5 Implementación del entorno

Para el tratamiento de las reglas de interpretación (evaluador RI) en las que interviene el entorno con variables y en las llamadas recursivas para el paso de parámetros por valor hemos utilizado una pila de entornos de variables.

Por otra parte, en otro entorno separado completamente del anterior, es donde disponemos de los cuerpos de las funciones declaradas por el usuario.

Esta separación llevada acabo en dos entornos es una cuestión de diseño y eficiencia, la razón es que no tiene sentido a la hora de realizar una llamada a una función realizar una copia de las funciones también en la pila del entorno de variables sobre el que se hace el paso de parámetros, con el consiguiente desaprovechamiento de memoria.

Tanto la creación de pila de entornos, como el entorno de funciones se han implementado con un patrón de creación ***Singleton***.

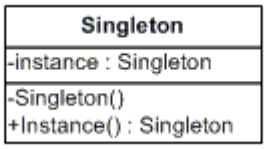


Figura 8.5: Patrón Singleton.

Este patrón nos asegura que sólo existe una instancia única de estas clases. *Singleton* define un método de clase que permite acceder a la instancia única y crearla en el caso de que no exista. El porqué de la utilización de este patrón es que no nos interesa tener más de una pila de entornos o un entorno de funciones en toda la aplicación, la razón es que son únicos y además son utilizados por distintos módulos del sistema.

9.6 Implementación del intérprete

La siguiente cuestión que abordamos es la implementación del intérprete. Es decir, implementar las acciones expuestas en el apartado correspondiente al intérprete (evaluador *RI*), así como transformar el árbol sintáctico de la gramática proporcionado por JavaCC de

una cadena de nuestro lenguaje, al árbol abstracto adecuado para su interpretación y visualización (traductor).

Estas acciones pueden implementarse como un recorrido de nuestro árbol, pero el problema principal es que el código correspondiente a la interpretación, visualización o transformación está disperso en cada una de las clases del árbol sintáctico o abstracto.

En la práctica, la construcción de un procesador de un lenguaje o un intérprete puede requerir la realización de varias fases: impresión, generación de código, interpretación, chequeo de tipos, etc. Y por tanto, si mantenemos el código en cada nodo del árbol estamos de alguna manera "ensuciando el código", y por tanto haciéndolo menos reutilizable, legible, susceptible a cambios, etc.

Mediante el patrón de comportamiento ***Visitor*** es posible concentrar el código de cada fase en una sola clase. Para ello, se define un método *accept*() en cada uno de los tipos de nodos del árbol (expresiones). Lo único que realiza dicho método es identificarse a sí mismo invocando el método correspondiente de la clase visitante, y de esta manera realizar el tratamiento adecuado.

Este patrón es el que nos ha permitido hacer los distintos recorridos del árbol abstracto para su visualización (estados relevantes del árbol abstracto en cada paso de la interpretación-evaluación) e interpretación-evaluación, así como para la transformación del árbol sintáctico proporcionado por JavaCC de una cadena determinada en nuestro código intermedio. Es decir, el árbol abstracto que podemos ver en *"cardida.model.interpreter.astnode"*, configuración del entorno, etc.

La aplicación del patrón *Visitor* se llevó acabo por los siguientes motivos:

- La estructura "árbol" contiene muchos tipos de nodos y es necesario llevar a cabo operaciones sobre estos nodos que son distintas unas de otras.
- Se quieren llevar a cabo operaciones dispares sobre estos nodos sin tener que incluir dichas operaciones en las clases. De esta manera podemos reutilizar la misma estructura árbol para distintas situaciones.
- Las clases que definen la estructura de "árbol" no cambian, pero las operaciones que se llevan a cabo sobre ellas sí.

La estructura y diagrama de secuencia del patrón *Visitor* es la siguiente:

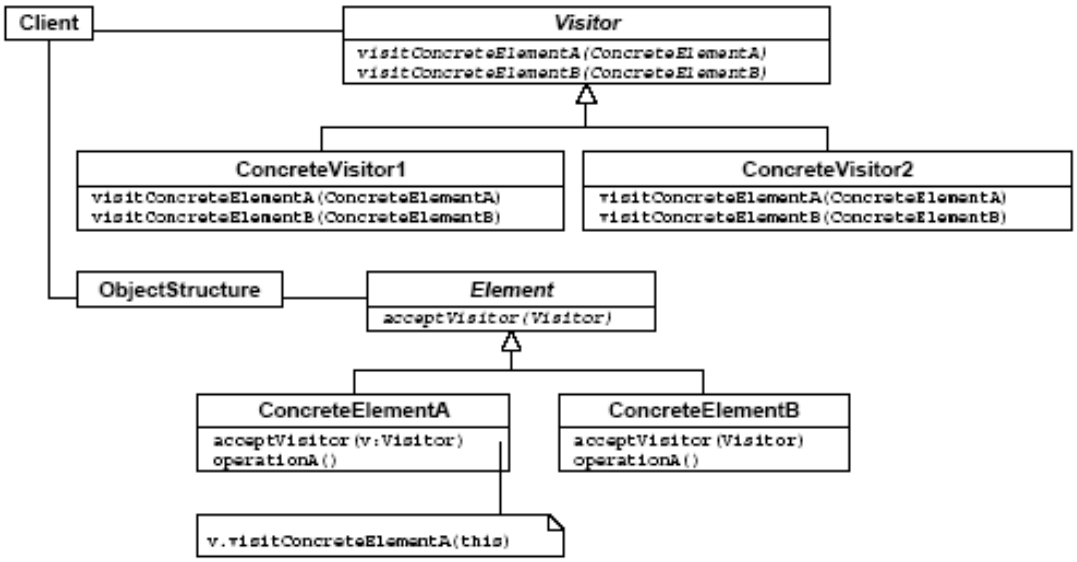


Figura 8.6: Diagrama de clases del patrón Visitor

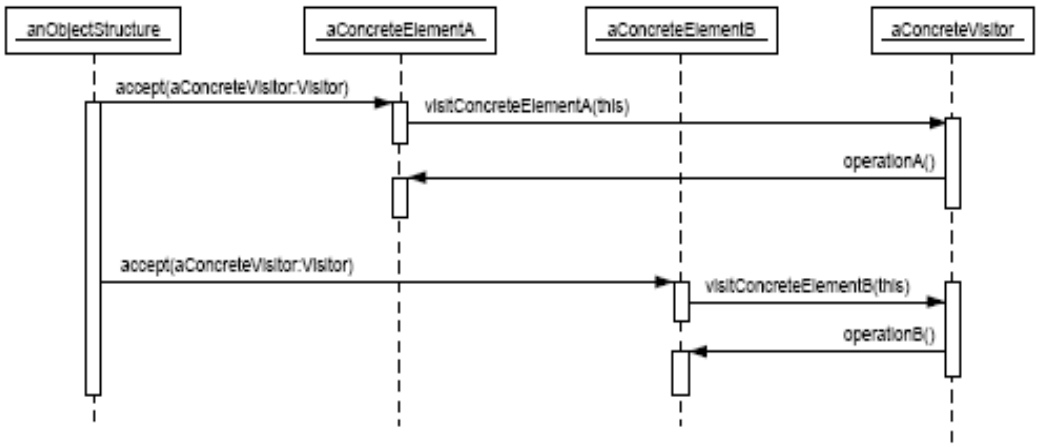


Figura 8.7: Diagrama de secuencia del patrón Visitor

Y sus participantes los siguientes:

- *Visitor*: Declara una operación de visita para cada uno de los elementos concretos de la estructura de objetos.
- *ConcreteVisitor*: Implementa cada una de las operaciones declaradas por Visitor. Normalmente, cada operación implementa una parte de la labor global del *Visitor* concreto, pudiendo almacenar información local.
- *Element*: Define la operación que le permite aceptar la visita de un *Visitor*.

- *ConcreteElement*: Implementa la operación de *accept*() que se limita a invocar su correspondiente método del *Visitor*.
- *ObjectStructure*: Gestiona la estructura de objetos y puede ofrecer una interfaz de alto nivel para permitir a los *Visitor* visitar a sus elementos.

Por tanto, utilizando este patrón, podemos definir el intérprete y el proceso de interpretación mediante las reglas definidas en el apartado 8.3, *Reglas para Eval* y en el 8.4, *Reglas para Apply*, como un posible recorrido del árbol y por tanto implementarlo mediante un *Visitor*.

Este patrón también es el utilizado en la visualización, así como el que da formato a la reescritura en un momento dado de la interpretación.

Para la creación de los distintos *Visitor* hemos utilizado también el patrón *Abstract Factory*, con las ventajas ya comentadas que ello conlleva.

En resumen el *Visitor* nos ha proporcionado lo siguiente:

- Todo el código del intérprete, de la visualización, de la reescritura, etc. está localizado en distintas clases facilitando así las modificaciones.

- Podrían añadirse nuevos tipos de recorridos como generación de código, chequeo de tipos, etc. de forma sencilla. Cada tipo de recorrido será una subclase de *Visitor*. Y al añadir un tipo de recorrido, no es necesario modificar el código del árbol sintáctico o el código del árbol abstracto.

Sin embargo, este diseño también tiene varias desventajas:

- Es más difícil añadir nuevos tipos de nodos al árbol sintáctico. Al hacerlo, habría que modificar todas las subclases de *Visitor*.
- Se debe exponer el interior de los nodos del árbol sintáctico, perjudicando la encapsulación.
- Todos los métodos del tipo "Object visitaX(X ...)" devuelven un valor Object obligando a realizar ahormados que pueden acarrear problemas de seguridad.

9.7 Generación de páginas html

Respecto al diseño de las páginas HTML generadas por *Cardida*, destacamos que han sido creadas mediante HTML Dinámico. Bajo el nombre de HTML Dinámico se engloba un conjunto de técnicas con dos objetivos claros: proporcionar un control absoluto al diseñador de páginas HTML y romper con el carácter estático de este tipo de documentos.

Un componente del HTML Dinámico son las *hojas de estilo*. Estas hojas de estilo han sido utilizadas porque permiten especificar atributos para los elementos de nuestra página Web.

Antes de la introducción de las hojas de estilo, los creadores de páginas Web sólo tenían un control parcial sobre el aspecto final de sus páginas. Por ejemplo, se podía

especificar que cierto texto debía verse como una cabecera, pero no se podían colocar márgenes a una página ni escoger un borde decorado para un texto.

Las hojas de estilo nos han permitido un mayor control sobre el aspecto de nuestros documentos. Con ellas hemos podido especificar muchos atributos tales como colores, márgenes, alineación de elementos, tipos y tamaños de letras, etc.

10 Conclusiones y trabajos futuros

Hemos investigado sobre la visualización del software, desarrollando un intérprete de visualizaciones capaz de visualizar un programa, cuyo lenguaje fuente hemos diseñado basándonos en lenguajes como Haskell o ML. El sistema Cardida desarrollado, además es capaz de generar animaciones Web. Estas animaciones se crean sin apenas esfuerzo por parte del usuario (profesor o alumno), minimizando así la interacción necesaria con el usuario para su construcción.

También se ha conseguido ver las dificultades que se presentan en el mundo de la visualización. Además hemos visto que la visualización de un programa funcional tiene una dificultad añadida más, ya que programa y datos son indivisibles. La razón es que los distintos resultados vienen expresados por el proceso de reescritura. Por último, también surge la dificultad de conseguir una expresividad visual efectiva.

En futuras versiones debería dotarse al sistema de una mayor expresividad visual, para ello la implementación de algoritmos de visualización se llevaría a cabo sin hacer uso de herramientas auxiliares como GraphViz. Por otro lado podría pensarse en la posibilidad de añadir a la aplicación la capacidad de mostrar árboles de recursividad, capacidad de controlar la evaluación, posibilidad de fijar focos de atención, comprobación de tipos, compartición de estructuras, implementación de evaluación perezosa, etc.

11 Interfaz gráfica y manual de usuario

El intérprete de visualización *Cardida* presenta una interfaz gráfica cuidada que facilita la interacción del usuario con el programa. Al arrancar el programa, el aspecto de la interfaz es el mostrado por la figura 9.1:

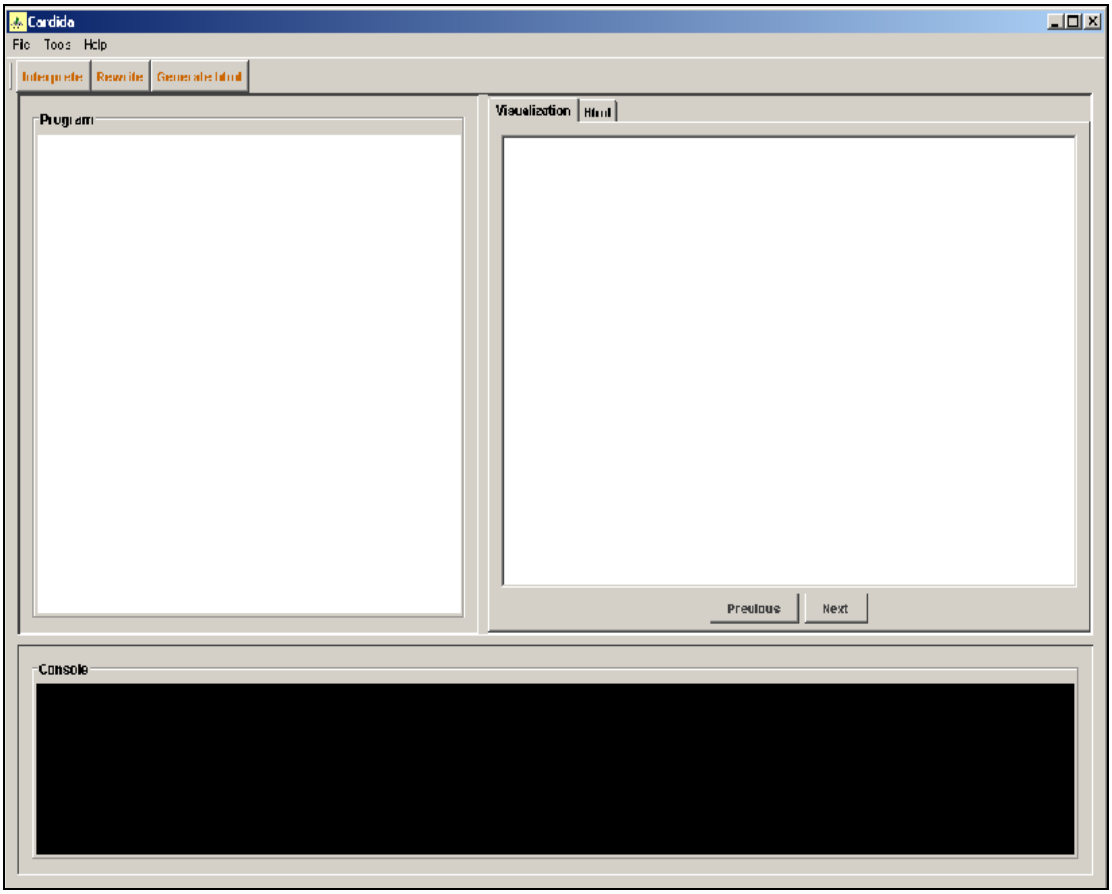


Figura 9.1: La interfaz principal

En dicha interfaz pueden apreciarse tres ventanas o paneles de texto distintos. El panel superior izquierdo está destinado al código del programa que se interpretará. En él se editará el código fuente del programa, bien editándolo a mano o bien cargando ese programa desde un archivo.

La ventana inferior está destinada a mostrar la reescritura del código. En ella se irán mostrando los sucesivos pasos de reescritura del programa a medida que este se va interpretando.

La interfaz principal de *Cardida* presenta tres pestañas principales: *File*, *Tools* y *Help* (Archivo, Herramientas y Ayuda respectivamente), y tres botones de interacción con el programa: *Interprete, Rewrite* y *Generate html* (Interpretar, Reescritura y Generación de html).

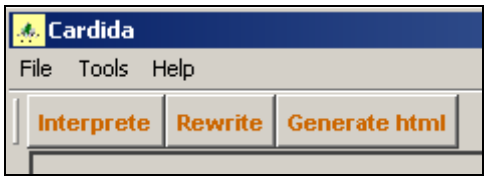


Figura 9.2: Menú de programa

Para comenzar a interactuar con la aplicación es necesario darle un programa que pueda tratar. Existen dos formas de hacerlo. La primera sería escribir directamente el código del programa que deseamos interpretar en la ventana superior izquierda. La segunda manera sería editar un programa ya existente.

Para editar un programa desde un archivo, pulsaremos sobre la pestaña *File,* que despliega un menú desde el que podemos seleccionar distintas opciones para crear o editar un programa.

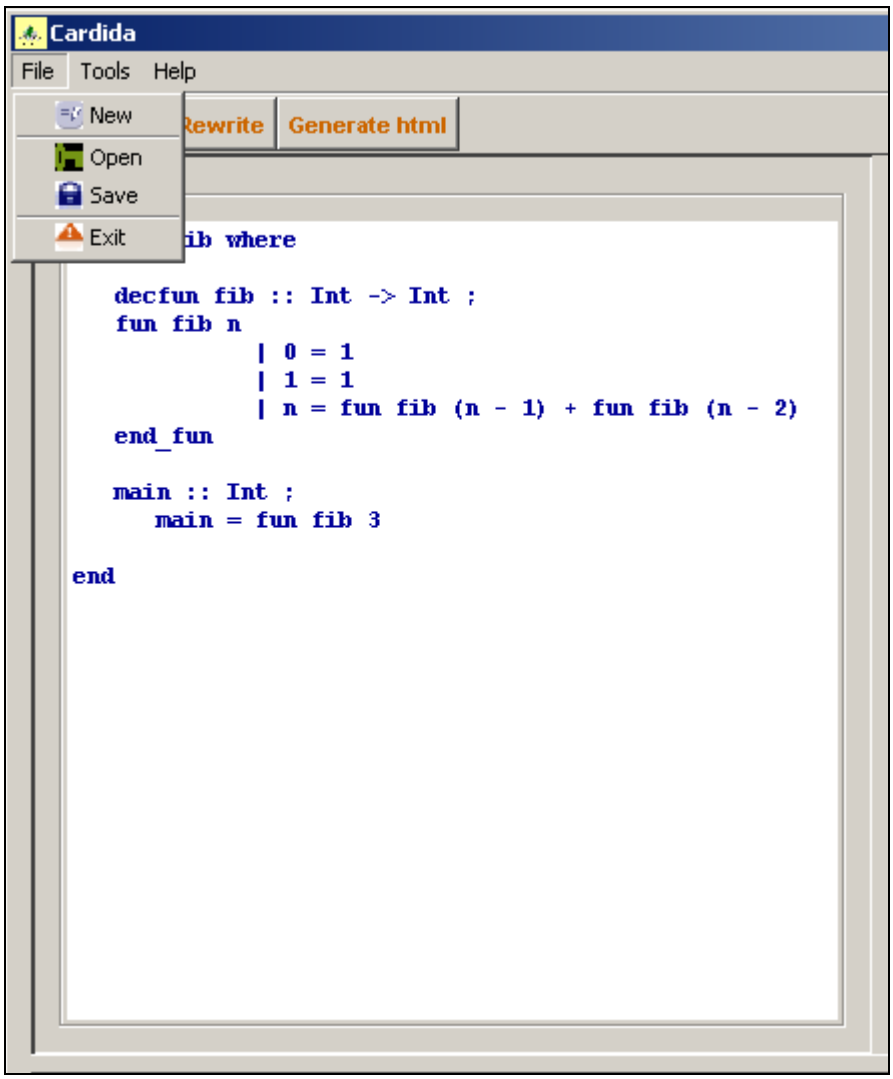


Figura 9.3: Opciones de la pestaña *File*

En la figura 9.3 pueden observarse las distintas opciones que ofrece el menú *File*. Seleccionar la opción *New* serviría para crear un nuevo programa desde cero, ya que limpia la ventana de edición de código de cualquier texto que tuviera escrita para prepararla para escribir en ella un nuevo programa.

La opción *Open* permite editar un archivo existente. Al seleccionar esta opción se abre una nueva ventana desde donde puede seleccionarse el programa a editar. El archivo debe llevar la extensión "**.cdd*", para ser reconocido por el intérprete. Por defecto el programa abre y guarda ficheros que tengan dicha extensión.

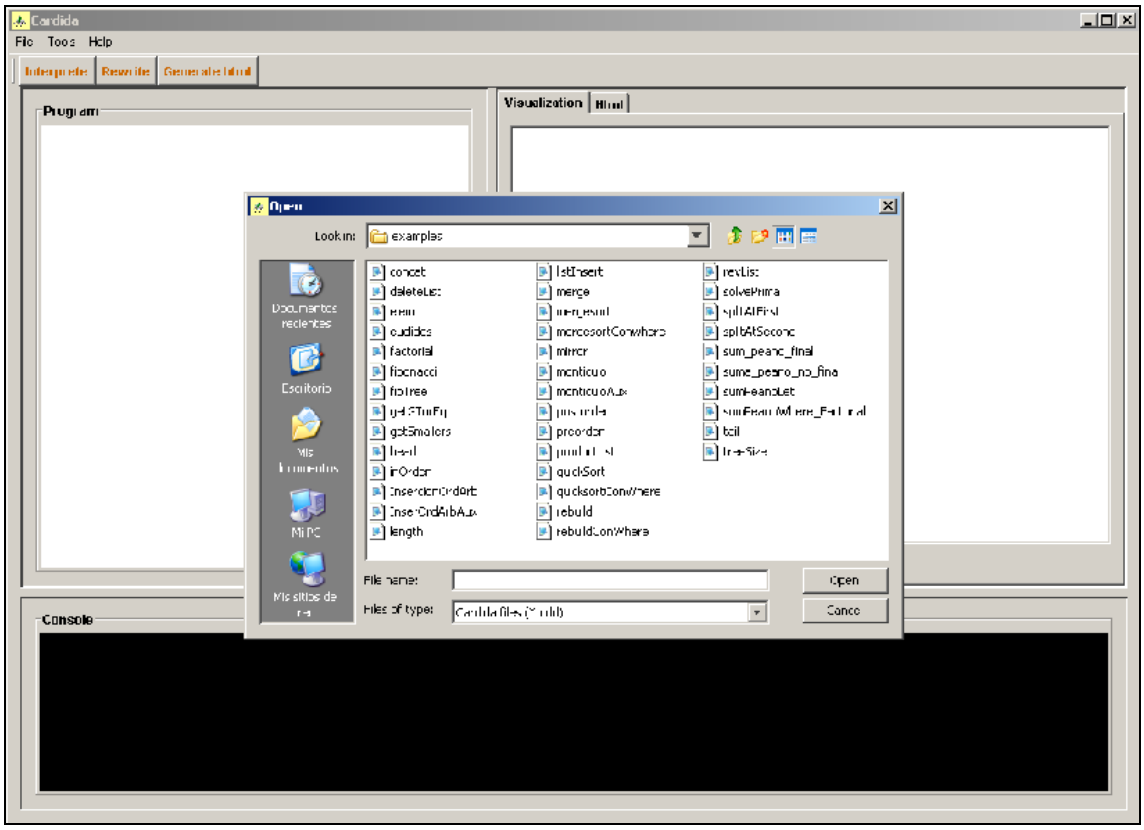


Figura 9.4: Editar archivos

Seleccionar un archivo hará que éste se cargue en el programa para ser interpretado, y que su código fuente aparezca en la ventana superior izquierda del panel principal.

La opción *save* guarda en un archivo, por defecto con la extensión *.cdd*, el programa que actualmente se encuentra en el editor de texto.

Seleccionar la función *exit* hará que el programa finalice y la interfaz de *cardida* se cierre.

En la pestaña *Tools* se despliegan las mismas opciones de reescritura, interpretación y generación de páginas html, además de la opción de establecer las características de la representación.

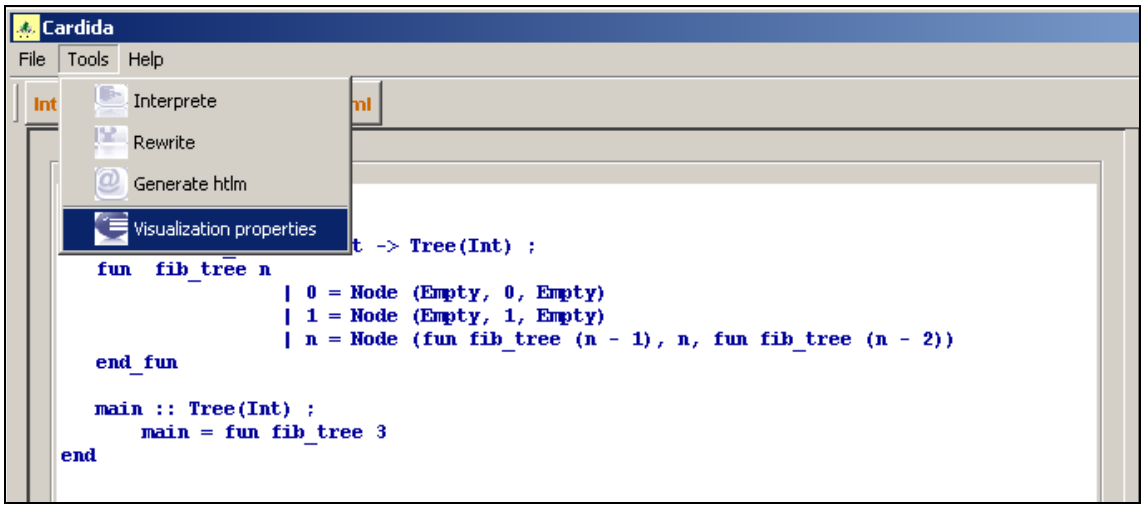


Figura 9.5: Menú de herramientas

A través de esta opción de personalización de las características de la representación, el usuario puede modificar aspectos relativos a los colores de las listas, los nodos de los árboles y de los dibujos mostrados.

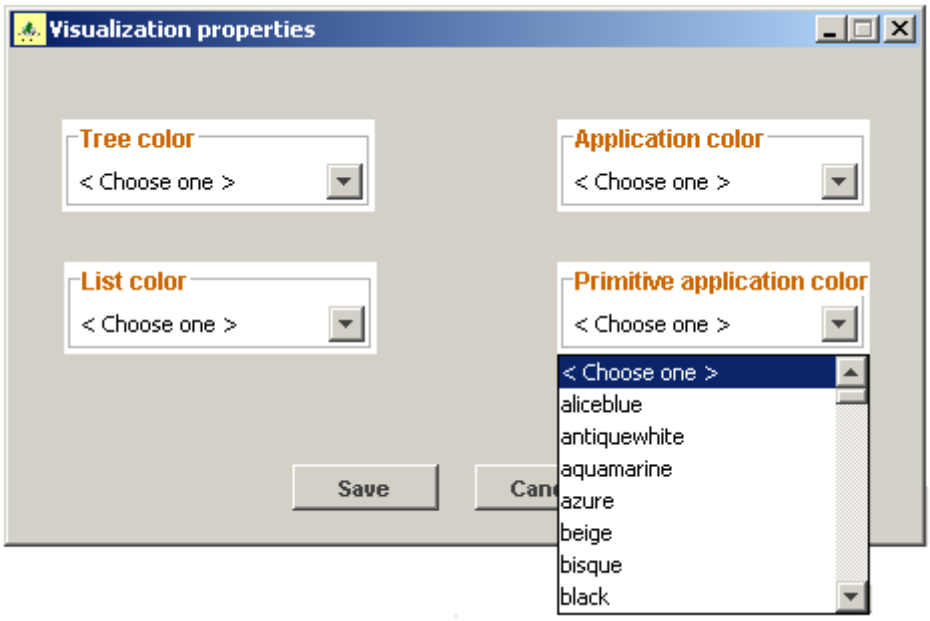


Figura 9.6: Selección de las propiedades de visualización

La interacción con el intérprete de visualizaciones puede hacerse mediante las opciones del menú de herramientas o mediante los botones de fácil acceso de la interfaz principal.

La opción de reescritura (*Rewrite*) produce la reescritura del código del programa fuente en la pantalla inferior del panel principal. El tamaño de esta ventana, al igual que las

otras dos, tiene un tamaño configurable, que se puede reajustar según convenga para observar mejor la reescritura del código.

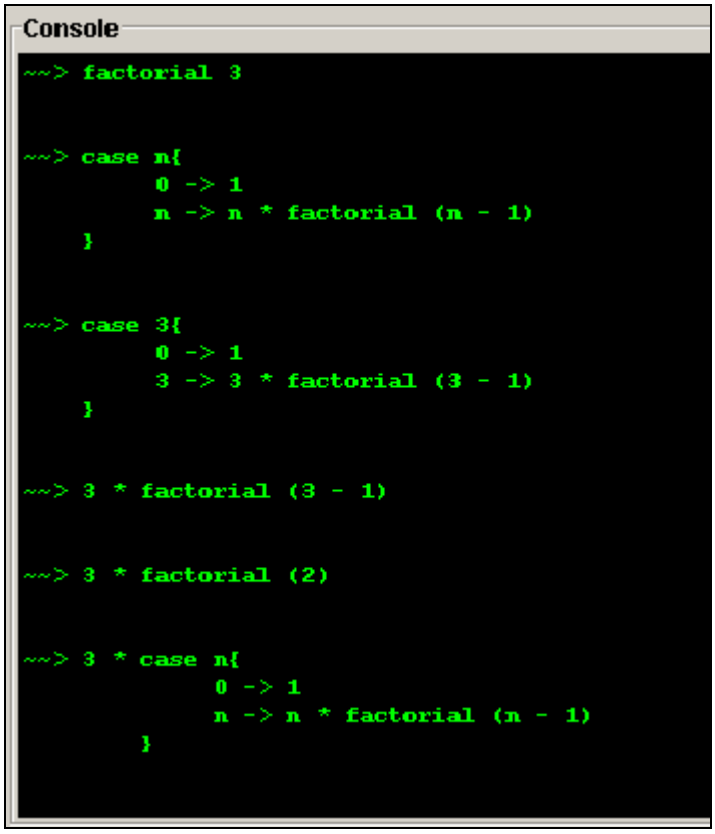


Figura 9.7: Reescritura de código fuente

La opción de interpretar (*Interprete*) produce además de la reescritura del código fuente, la representación del mismo en la ventana superior derecha. En dicha ventana, en la pestaña nombrada *Visualización*, aparecerá la primera imagen correspondiente a la representación del programa interpretado. A través de los botones *Previous* y *Next* (anterior y siguiente, respectivamente) podemos manipular la representación para mostrar visualización de cada paso de la evolución del intérprete.

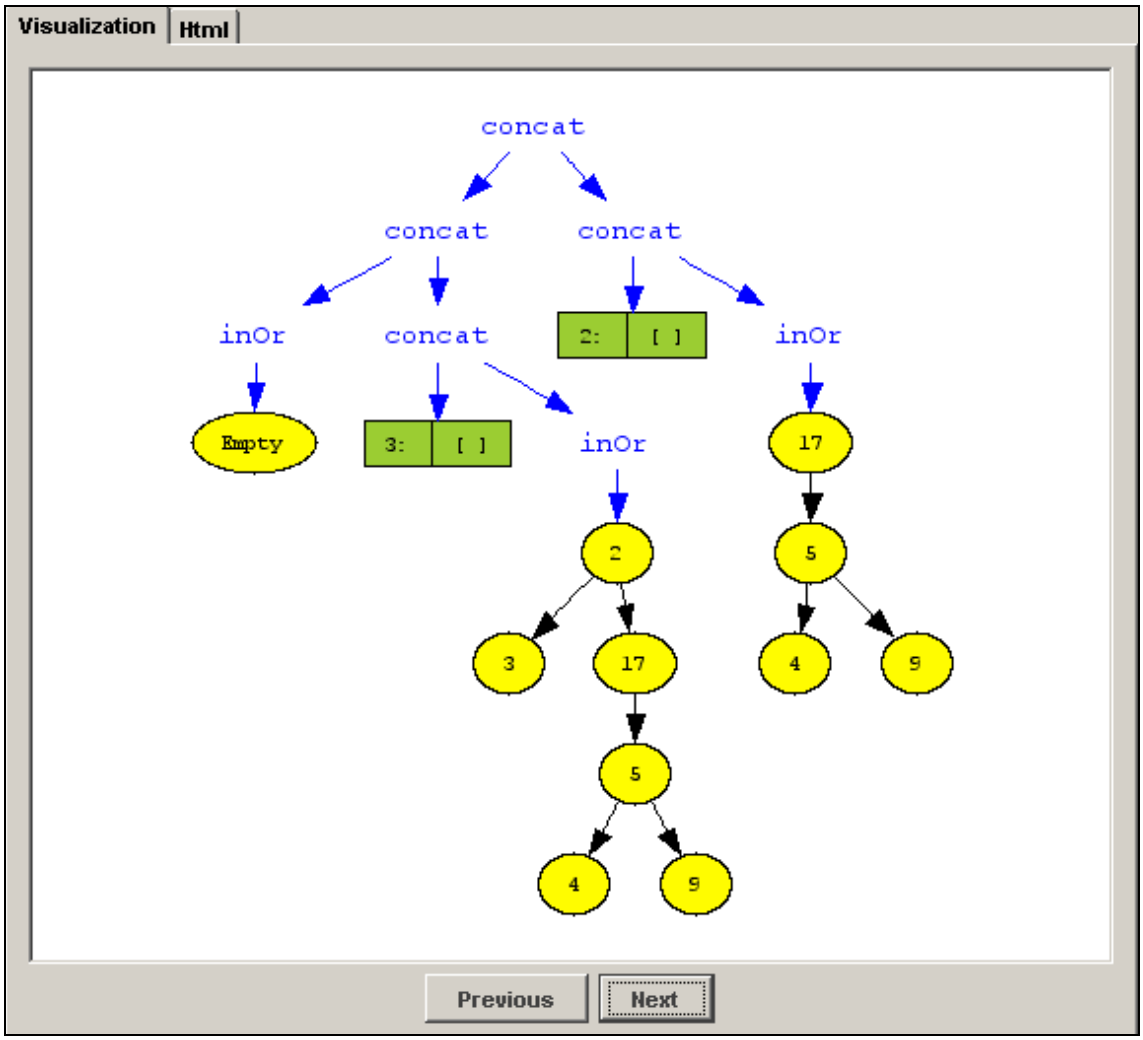


Figura 9.8: Representación de la ejecución de un programa

El tercer botón de la interfaz del intérprete de visualización genera páginas html con las imágenes de la representación del programa fuente. Para generar dichas páginas adecuadamente, se puede introducir texto en ellas a través de los campos de la pestaña nombrada como *html* en la ventana superior derecha de la pantalla principal. De esta forma se puede especificar un título para el programa que se presentará en la página, una descripción del problema a resolver y una descripción del algoritmo usado para resolverlo.

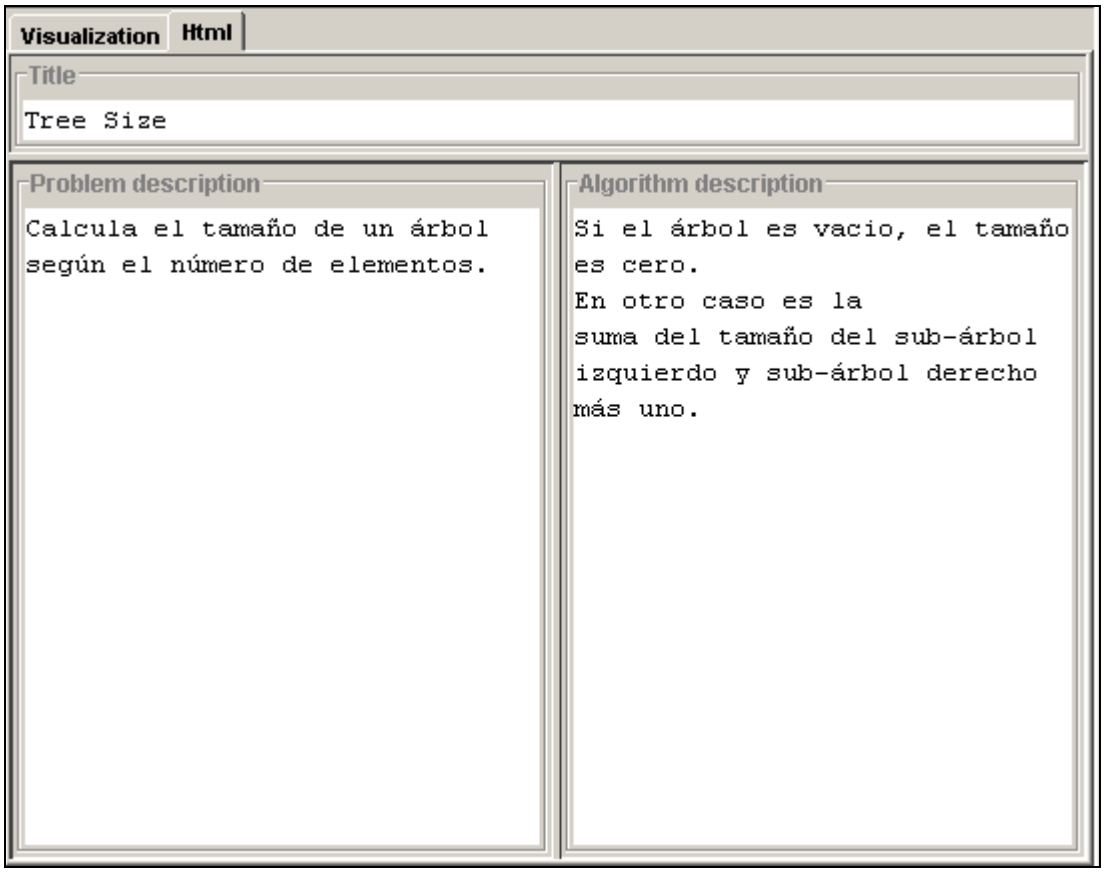


Figura 9.9: Generación de páginas html

12 Colección de ejemplos

Los autores de este proyecto, hemos tratado de incluir una colección de ejemplos amplia para facilitar la interacción con el intérprete de visualizaciones. El objetivo es presentar una serie de programas representativa de las características más relevantes del mismo. De esta forma pueden observarse una variedad de algoritmos y programas básicos en el paradigma funcional, como los del tipo de manipulación de listas o árboles, sin necesidad de que el usuario tenga que programarlos desde cero.

Además de proporcionar ya implementados los problemas clásicos, proporcionar una serie de ejemplos ayudará a los usuarios de la aplicación a crear sus propios programas, ya que poseerán una amplia colección de ejemplos como referencia para corregir los problemas de sus programas o para guiar el desarrollo de los suyos propios.

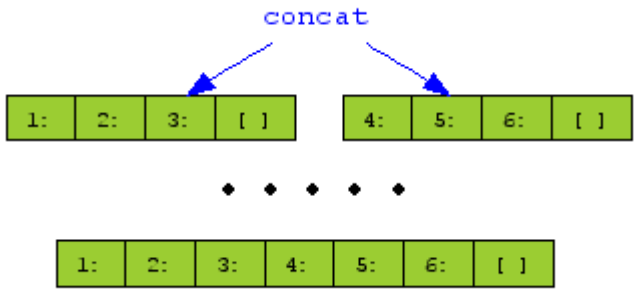
Esta colección sería de especial ayuda a los usuarios que sean alumnos que usen la aplicación para aprendizaje, ya que podrán empezar a interaccionar con el intérprete a través de la colección de ejemplos, y pasar luego a experimentar con sus propios programas cuando alcancen un grado mayor de confianza con la aplicación.

A continuación describiremos cada uno de los ejemplos proporcionados[2]:

1. concat.cdd

Descripción del problema: Concatenación de dos listas.

Descripción del algoritmo: Dadas dos listas, si la primera de ellas es vacía, el resultado de la concatenación es la segunda lista. Si la primera lista está formada por cabeza y cola, el resultado es una lista que tiene por cabeza: la cabeza de la primera lista; y como cola: la concatenación de la cola de la primera lista con la segunda lista.



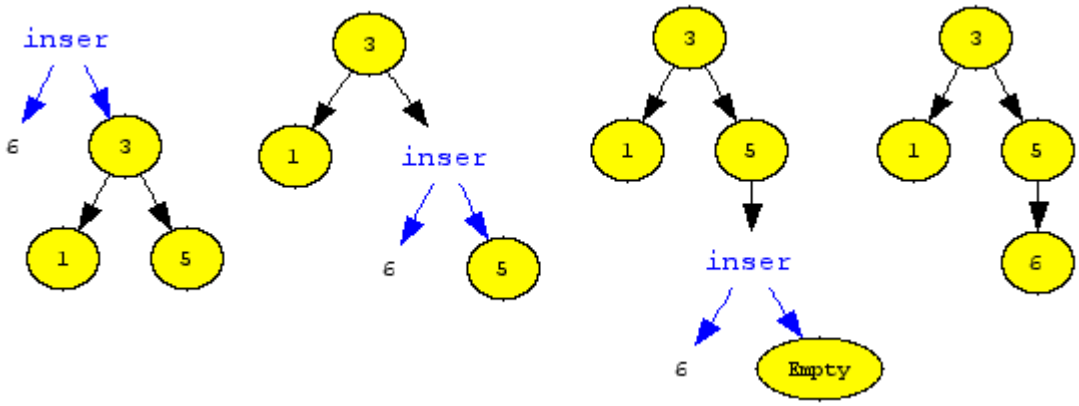
Ejemplo de concatenación de listas

[2] Puede consultarse una colección de ejemplos de Cardida en http://aljibe.sip.ucm.es/cardida/ , así como en el cd adjuntado.

2. inserOrdArb.cdd

Descripción del problema: Insertar en un árbol ordenado.

Descripción del algoritmo: Si el elemento a insertar es más pequeño que la raíz del árbol se inserta en el subárbol izquierdo. Si es mayor se inserta en el subárbol derecho. Si es igual no se inserta, para no contemplar números repetidos. Si el árbol es vacío, el elemento a insertar es la raíz de un nuevo árbol cuyos dos hijos son vacíos.

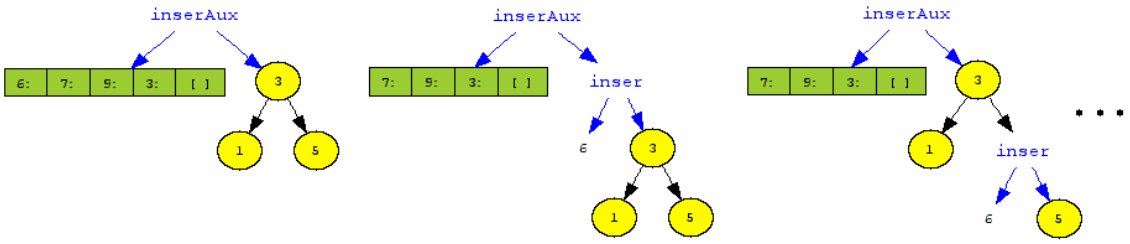


Ejemplo de inserción ordenada en un árbol

3. inserOrdArbAux.cdd

Descripción del problema: Insertar en un árbol ordenado elementos de una lista

Descripción del algoritmo: Si el elemento de la lista a insertar es más pequeño que la raíz del árbol se inserta en el subárbol izquierdo. Si es mayor se inserta en el subárbol derecho. Si es igual no se inserta, para no contemplar números repetidos. Si el árbol es vacío el elemento a insertar es la raíz de un nuevo árbol cuyos dos hijos son vacíos. Así es procesado cada elemento de la lista.



Ejemplo de inserción de los elementos de una lista

4. factorial.cdd

Descripción del problema: El factorial de un número n es una función recursiva de la forma: factorial(n) = n*factorial(n-1).

Descripción del algoritmo: El caso recursivo es factorial(n) = n*factorial(n-1), y el caso base es factorial(0)=1.

```
factorial 3

3 * factorial (3 - 1)

3 * factorial (2)

3 * 2 * factorial (2 - 1)

3 * 2 * factorial (1)

3 * 2 * 1 * factorial (1 - 1)

3 * 2 * 1 * factorial (0)

3 * 2 * 1 * 1

3 * 2 * 1

3 * 2

6
```

Ejemplo de factorial

5. fibonacci.cdd

Descripción del problema: Cálculo del n-ésimo número de Fibonacci.

Descripción del algoritmo: El n-ésimo número de Fibonacci es la suma de los dos anteriores. Donde los dos primeros números de la sucesión son: Fibonacci(0) y Fibonacci(1), de valor uno.

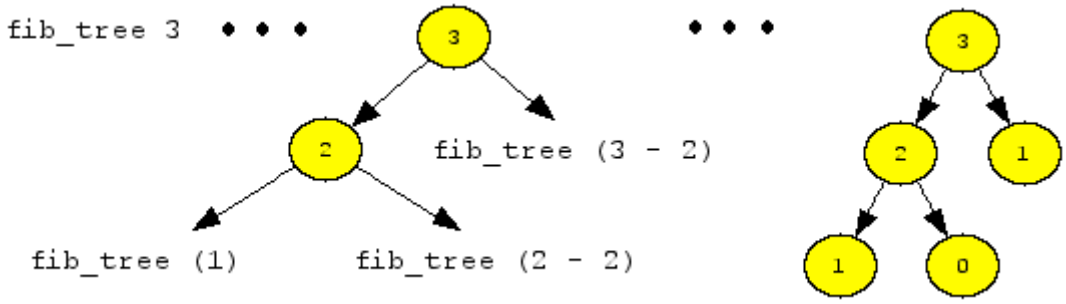
```
fib 2

fib (2 - 1)  + fib (2 - 2)

fib (1)  + fib (2 - 2)

1 + fib (2 - 2)

1 + fib (0)

1 + 1

2
```

Ejemplo de fibonacci

6. fibTree.cdd

Descripción del problema: Árbol de llamadas creado en el cálculo del n-ésimo número de Fibonacci.

Descripción del algoritmo: El caso recursivo, es un árbol donde la raíz es el parámetro de llamada de fibonacci y cuyos hijos son los árboles creados por fibonacci(n-1) y fibonacci(n-2). Los casos base son árboles que sólo tienen raíz cero y uno, y que se corresponden con los primeros términos de la sucesión de Fibonacci (fibonacci(0) y fibonacci(1)).

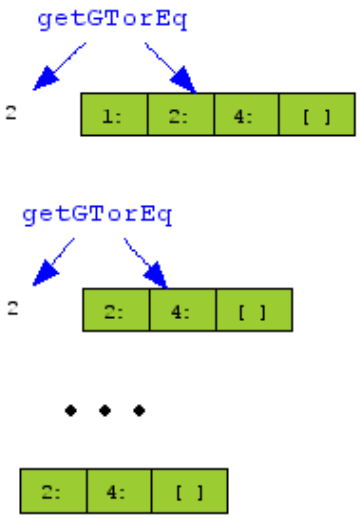


Ejemplo del árbol creado por fibonacci

7. getGTorEQ.cdd

Descripción del problema: Cálculo de los elementos de una lista mayores o iguales a uno dado.

Descripción del algoritmo: Si la cabeza de la lista es menor o igual que el elemento de comparación se concatena con la lista resultante de calcular los elementos menores o iguales del resto de la lista. Si es mayor, se aplica recursivamente el algoritmo sobre la lista sin la cabeza.

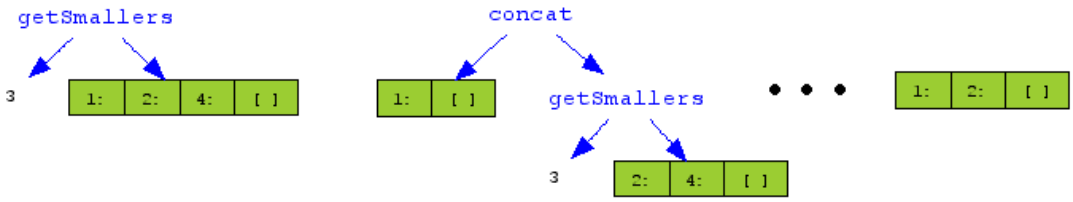


Ejemplo de la función getGTorEQ

8. getSmallers.cdd

Descripción del problema: Cálculo de los elementos de una lista menores que uno dado.

Descripción del algoritmo: Si la cabeza de la lista es menor que el elemento de comparación, se devuelve la lista resultante de concatenar la cabeza con la lista de los elementos menores o iguales del resto de la lista. Si es mayor o igual, se aplica recursivamente el algoritmo sobre el resto de la lista.

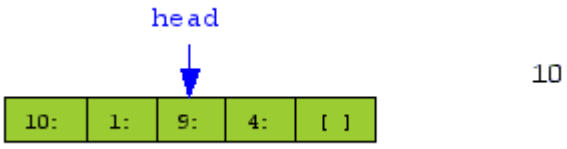


Ejemplo de la función getSmallers

9. head.cdd

Descripción del problema: Calcula la cabeza de una lista.

Descripción del algoritmo: Se usa una función muy simple, que dada una lista formada por cabeza y cola devuelve la cabeza.



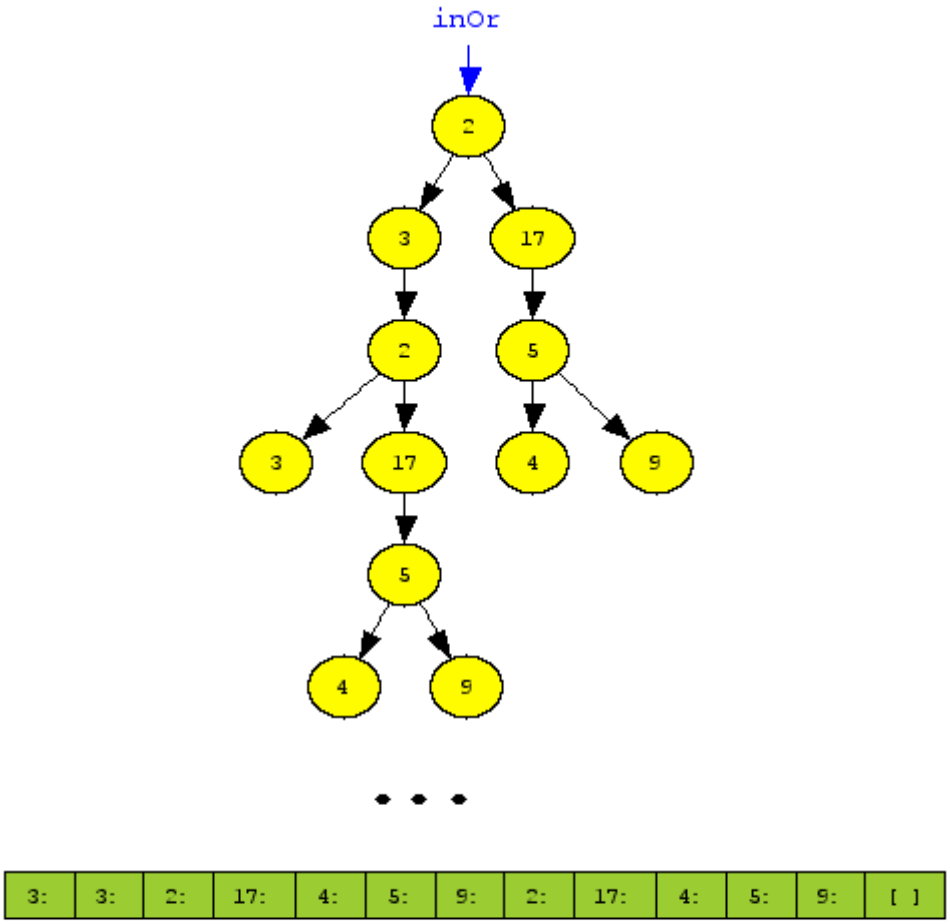
Ejemplo de la función head

10. inOrden.cdd

Descripción del problema: Dado un árbol de elementos, se devuelve la lista de elementos resultante de hacer el recorrido en inorden del árbol.

Descripción del algoritmo: El recorrido en inorden de un árbol consiste en recorrer en inorden el subárbol izquierdo, recorrer la raíz, y recorrer en inorden el subárbol derecho.

Se mantiene la función auxiliar de concatenar dos listas, por lo que la manera de obtener una lista con el recorrido en inorden del árbol es concatenar la lista resultante de recorrer en inorden el hijo izquierdo y la concatenación de la lista unitaria del elemento raíz con el recorrido en inorden del hijo derecho. El caso base es el recorrido de un árbol vacío, que devuelve la lista vacía.

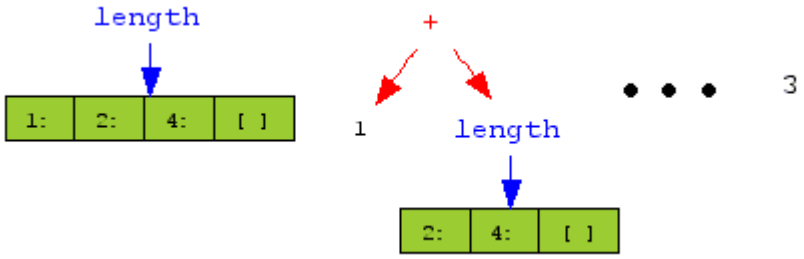


Ejemplo de inorden de un árbol

11. length.cdd

Descripción del problema: Calcula el número de elementos de una lista.

Descripción del algoritmo: Como caso base tenemos la longitud de la lista vacía, que por supuesto es cero. El caso recursivo es el de la lista formada por cabeza y cuerpo, donde la longitud de dicha lista es 1, por el elemento cabeza, más la longitud del cuerpo de la lista.

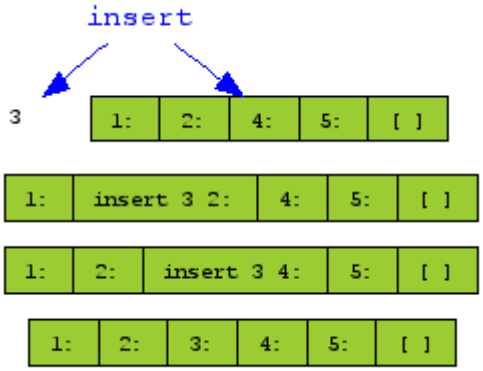


Ejemplo de la función length

12. listInsert.cdd

Descripción del problema: Inserta un elemento en una lista ordenada.

Descripción del algoritmo: Si la lista en la que queremos insertar es vacía, devolvemos la lista unitaria del elemento a insertar. Si la lista no es vacía, distinguiremos casos para ver si el elemento a insertar es mayor o menor que la cabeza de la lista. Si el elemento es menor que la cabeza de la lista, este pasa a ser la nueva cabeza de lista. Si es mayor o igual, se inserta en la cola de la lista.

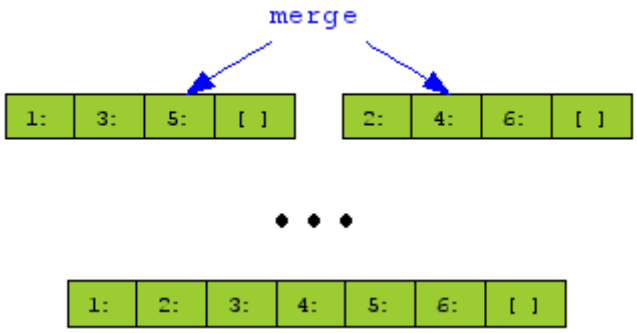


Ejemplo de inserción en una lista ordenada

13. merge.cdd

Descripción del problema: Mezcla de dos listas ordenadas

Descripción del algoritmo: *Merge* se usa para unir dos listas ordenadas en una sola. Para ello, esta función compara las cabezas de ambas listas y devuelve una lista cuya cabeza es la menor de las cabezas resultantes y con el cuerpo resultante de llamada recursiva a la función con el cuerpo de la lista que tenía menor cabeza y la otra lista.



Ejemplo de la función merge

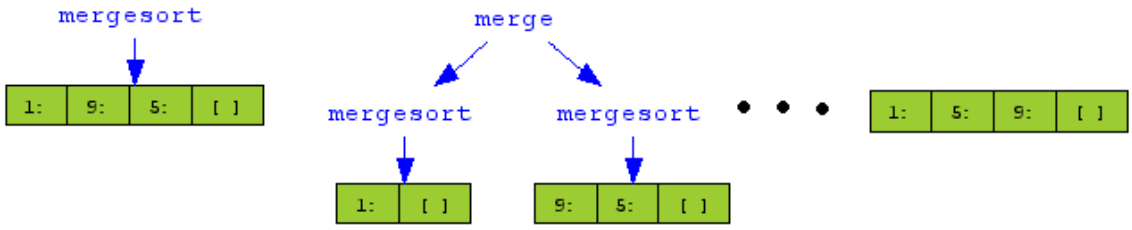
14. mergesort.cdd y mergesortConWhere.cdd

Descripción del problema: Algoritmo de ordenación por mezcla.

Descripción del algoritmo: El algoritmo mergesort usa una estrategia tipo divide y vencerás para ordenar los elementos de una lista. Los casos base son la ordenación de una lista vacía o de una lista unitaria, que son listas ordenadas. Para ordenar una lista más grande se divide en dos partes que se ordenan de forma recursiva y a continuación se unen los elementos de dichas listas de una forma ordenada.

Para la implementación del algoritmo son necesarias funciones auxiliares. Las funciones *splitAtFirst* y *splitAtSecond* toman un entero n y una lista y devuelven respectivamente una lista con los "n" primeros elementos de la lista original, o la lista resultante de eliminar los "n" primeros elementos de la lista. Estas dos funciones sirven para dividir en dos partes iguales la lista original. La función *length* que calcula el número de elementos de una lista, es necesaria para poder dividir la lista en partes iguales.

La función auxiliar *merge* se usa para unir las dos listas ordenadas en una sola lista.

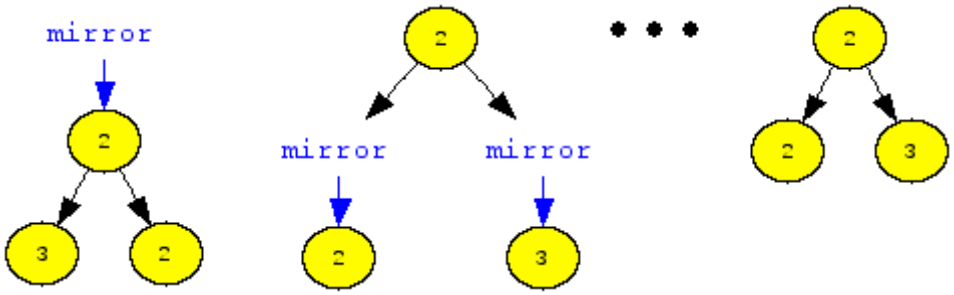


Ejemplo de ordenación mergesort

15. mirror.cdd

Descripción del problema: Calcula la imagen especular de un árbol, esto es, con un eje vertical de simetría centrado en la raíz.

Descripción del algoritmo: Si un árbol es vacío, su imagen especular es un árbol vacío. Si el árbol no es vacío, su imagen especular es la que resulta de construir un árbol cuyo hijo izquierdo es la imagen especular del hijo derecho, y cuyo hijo derecho también es la imagen especular del hijo izquierdo. La raíz no varía.

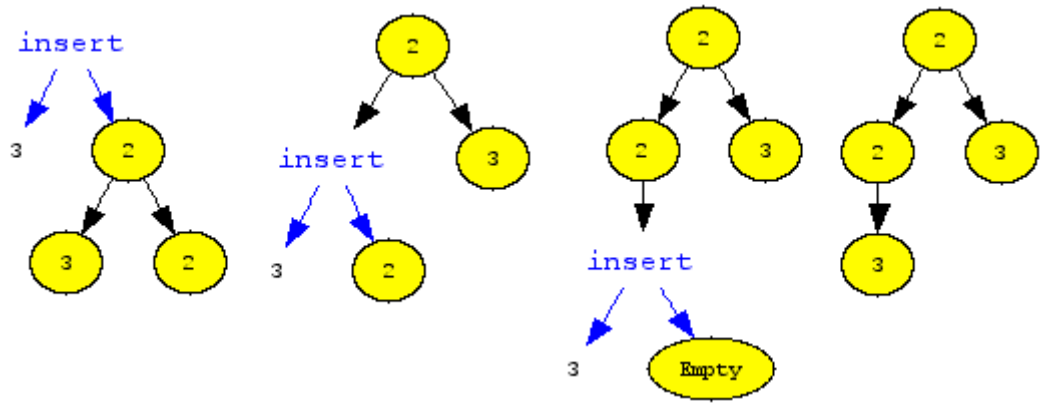


Ejemplo de la imagen especular de un árbol

16. monticulo.cdd

Descripción del problema: Inserción en montículo.

Descripción del algoritmo: Si el montículo es vacío, tenemos el caso trivial, creamos un montículo de un elemento. En cualquier otro caso, insertamos recursivamente el elemento en submontículo izquierdo.

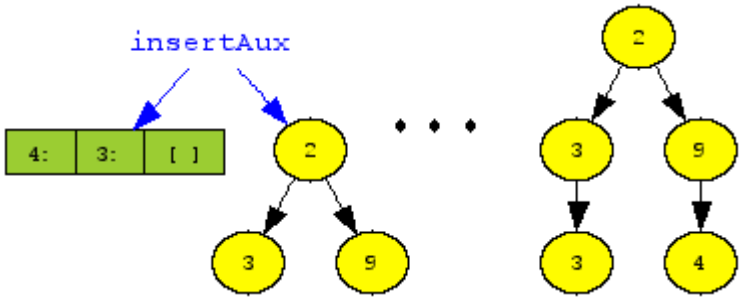


Ejemplo de inserción en un montículo

17. monticuloAux.cdd

Descripción del problema: Inserta una lista de elementos en un montículo.

Descripción del algoritmo: Si la lista es vacía devolvemos el montículo correspondiente, de lo contrario vamos insertando cada elemento de la lista de la en el montículo.



Ejemplo de inserción de elementos de una lista en un montículo

18. postOrden.cdd

Descripción del problema: Calcula el recorrido en postorden de un árbol.

Descripción del algoritmo: El recorrido en postorden de un árbol se realiza recorriendo en postorden el subárbol izquierdo, en postorden el subárbol derecho y por último la raíz. El caso base de este algoritmo es tratar de recorrer un árbol vacío, caso en el que se devuelve la lista vacía. Si el árbol no es vacío, el resultado es la concatenación del recorrido en postorden del hijo izquierdo, el recorrido en postorden del hijo derecho y la lista unitaria formada por el elemento raíz.

Ejemplo de postorden de un árbol

19. preorden.cdd

Descripción del problema: Calcula el recorrido en preorden de un árbol.

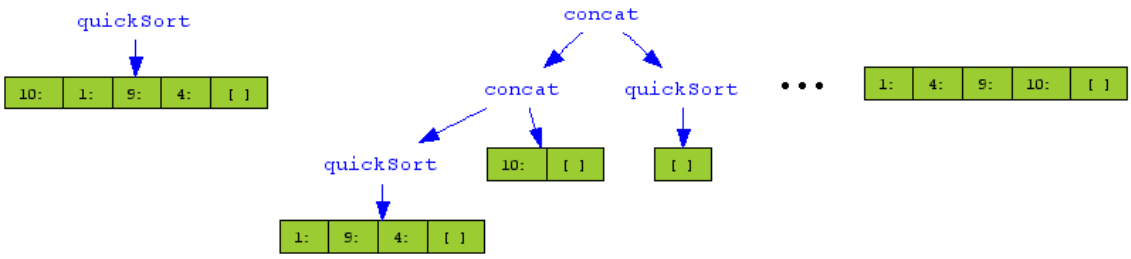
Descripción del algoritmo: El recorrido en preorden de un árbol se realiza recorriendo primero el elemento raíz, recorriendo en preorden el subárbol izquierdo y por último recorriendo en preorden el hijo el derecho. El recorrido de un árbol vacío devuelve una lista vacía. Si el árbol es no vacío se devuelve la concatenación de la lista unitaria del elemento raíz con las listas resultantes del recorrido en preorden de los subárboles izquierdo y derecho.

Ejemplo de preorden de un árbol

20. quicksort.cdd y quicksortConwhere.cdd

Descripción del problema: El algoritmo de ordenación quicksort.

Descripción del algoritmo: Este algoritmo de ordenación sigue una estrategia de tipo divide y vencerás. Se apoya en las funciones auxiliares ya explicadas anteriormente *getGTorEQ* y *getSmallers* para dividir la lista original en dos listas, con los elementos mayores y menores, respectivamente. Luego estas listas son ordenadas recursivamente de nuevo con *quicksort*, hasta llegar a un caso base que es la ordenación de la lista vacía. Las listas resultantes ya ordenadas se concatenan para obtener la lista final.



Ejemplo de ordenación quicksort

21. revList.cdd

Descripción del problema: Calcula la inversa de una lista, de forma que intercambian posiciones el primer y último elemento, el segundo y el penúltimo, y así sucesivamente.

Descripción del algoritmo: El caso base del algoritmo es una lista vacía, que es su propia inversa. Para una lista formada por cabeza y cola, la lista resultante es la concatenación de la inversa de la cola con la lista unitaria del elemento cabeza.

Ejemplo de inversa de una lista

22. splitAtFirst.cdd

Descripción del problema: Devuelve una lista con los n primeros elementos de la lista original.

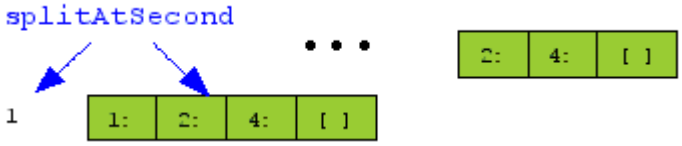
Descripción del algoritmo: Esta función recibe un entero y una lista. Si el entero que indica el número de elementos que se quieren tomar de una lista es cero, o la lista es vacía, se devuelve la lista vacía. El caso recursivo devuelve una lista formada por la cabeza original de la lista y como cuerpo la lista resultante de la llamada recursiva a la función *splitAtFirst* con el número de elementos a tomar menos uno.

Ejemplo de la función splitAtFirst

23. splitAtSecond.cdd

Descripción del problema: Devuelve una lista en la que se eliniman los n primeros elementos de la lista original.

Descripción del algoritmo: Esta función recibe un entero que indica el número de elementos a eliminar de la lista y una lista. Si el número de elementos que deseamos eliminar es cero, devolvemos la lista original. Si la lista es vacía, se devuelve la lista vacía. Si no lo es, si el número de elementos que deseamos eliminar es mayor que cero, devolvemos la lista resultante de la llamada recursiva a la función con el cuerpo de la lista y el número de elementos a eliminar menos uno.



Ejemplo de la función splitAtSecond

24. suma_peano_final.cdd

Descripción del problema: Implementación de la suma siguiendo los axiomas de Peano.

Descripción del algoritmo: Esta función calcula la suma de dos números. El caso base es aquel en el que el primer sumando es cero, y por tanto la suma es el valor del segundo operando. Si el primer número es mayor que cero, la suma se calcula de forma

recursiva como la suma de Peano del predecesor del primer operando y el sucesor del segundo.

```
peano 2 1

peano (2 - 1) (1 + 1)

peano (1) (1 + 1)

peano (1) (2)

  • • •

peano (0) (2 + 1)

peano (0) (3)

3
```

Ejemplo de la suma de Peano final

25. suma_peano_no_final.cdd

Descripción del problema: Implementación de la función de suma, siguiendo los axiomas de Peano.

Descripción del algoritmo: Este algoritmo es parecido al anterior, pero con una recursión no final. En este caso el valor devuelto es el sucesor de la llamada recursiva a la suma de Peano con el predecesor del primer elemento y el segundo elemento original. El caso base es que el primer elemento sea cero, en cuyo caso la suma es el valor del segundo elemento.

```
peano 4 2

1 + peano (4 - 1) 2

1 + peano (3) 2

1 + 1 + peano (3 - 1) 2

  • • •

1 + 1 + 1 + 1 + peano (1 - 1) 2

1 + 1 + 1 + 3

  • • •

6
```

Ejemplo de suma de Peano no final

26. tail.cdd

Descripción del problema: Dada una lista formada por cabeza y cola, devuelve la cola de la lista.

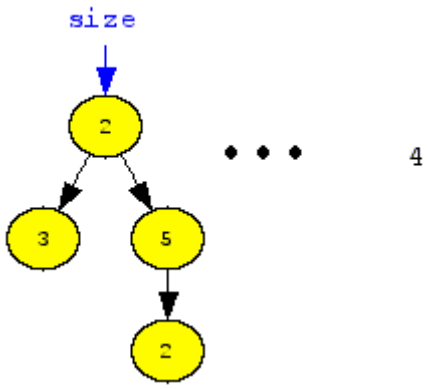
Descripción del algoritmo: Si la lista es vacía, el resultado es una lista vacía. Si no lo es, el resultado es el resto de la lista.

Ejemplo de la función tail

27. treeSize.cdd

Descripción del problema: Calcula el tamaño de un árbol.

Descripción del algoritmo: Si el árbol es vacío, su tamaño es cero. Si no lo es, su tamaño es uno (por el elemento raíz) más el tamaño de los subárboles izquierdo y derecho.



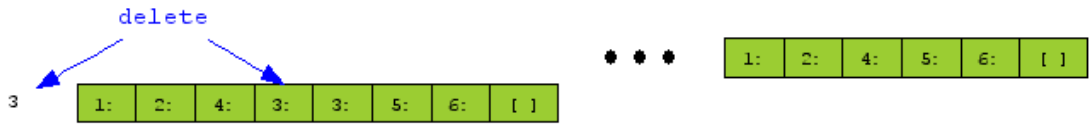
Ejemplo de tamaño de un árb0l

28. deleteList.cdd

Descripción del problema: Borrar un elemento de una lista.

Descripción del algoritmo: Si la lista es vacía devolvemos la lista (no hay que eliminar ningún elemento), de lo contrario recorremos la lista recursivamente: si la cabeza de la lista es igual al elemento buscado (el elemento que se quiere eliminar) entonces devolvemos la lista resultante de eliminar recursivamente el elemento de su resto, de lo

contrario devolvemos la lista formada por esa misma cabeza y con el resto resultante de la eliminación del elemento buscado.

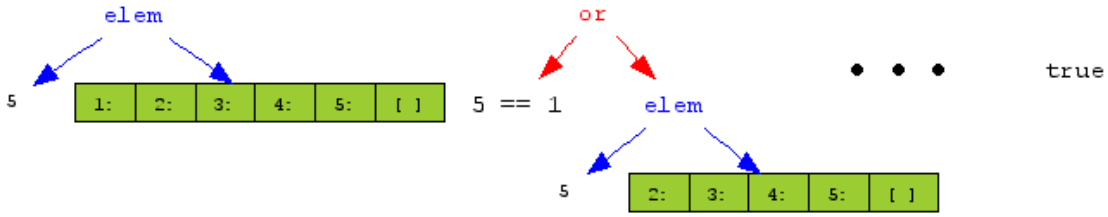


Ejemplo de la función delete

29. elem.cdd

Descripción del problema: Decide si existe un elemento en la lista.

Descripción del algoritmo: Si la lista es vacía, el elemento no se encontrará en la lista y por tanto se devuelve falso. De lo contrario, el elemento se encontrará en la lista si es igual a la cabeza o se encuentra en el resto de la lista.

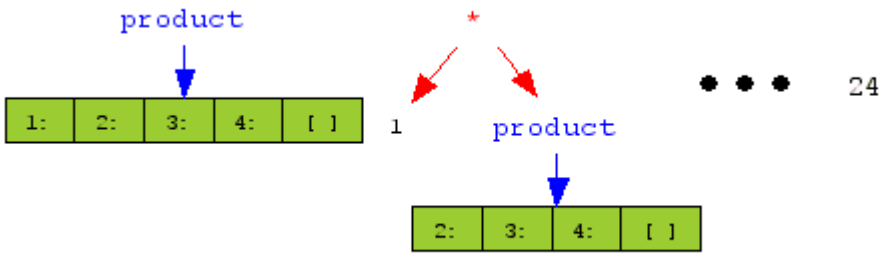


Ejemplo de la función elem

30. productList.cdd

Descripción del problema: Producto de los elementos de una la lista.

Descripción del algoritmo: Si la lista es vacía, inicializamos el producto a uno. De lo contrario, devolvemos el producto resultante de multiplicar la cabeza por el producto del resto de la lista.



Ejemplo de la función product

31. rebuild.cdd y rebuildConWhere.cdd

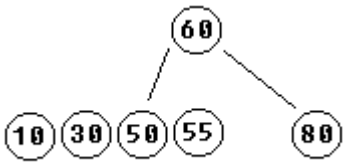
Descripción del problema: Construir un árbol a partir de su recorrido en preorden e inorden.

preorden: raiz , izquierdo , derecho
inorden: izquierdo , raiz , derecho

Descripción del algoritmo: A partir de los conceptos de raíz, subárbol izquierdo y derecho, se van identificando las raíces de los subárboles.

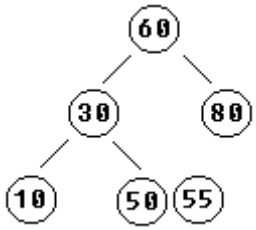
Por ejemplo si tenemos que los recorridos de un árbol binario (no necesariamente ordenado) son:

preorden: **60** - 30 - 10 - 50 - 55 - 80
inorden: 10 - 30 - 50 - 55 - **60** - 80



Según el recorrido preorden podemos identificar que el elemento 60 es raíz de un subárbol. Por lo tanto si nos fijamos en el recorrido inorden, vemos que:

subárbol izquierdo: 10 - 30 - 50 -55
subárbol derecho: 80

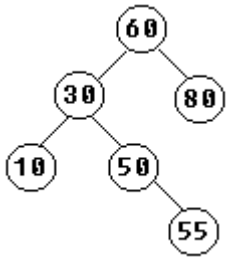


Luego, se dividen los recorridos de los distintos subárboles. Como en este caso existe sólo un subárbol izquierdo, se tiene:

preorden: **30** - 10 - 50 - 55
inorden: 10 - **30** - 50 - 55

Observando los recorridos se concluye que:

subárbol izquierdo: 10
subárbol derecho: 50 - 55

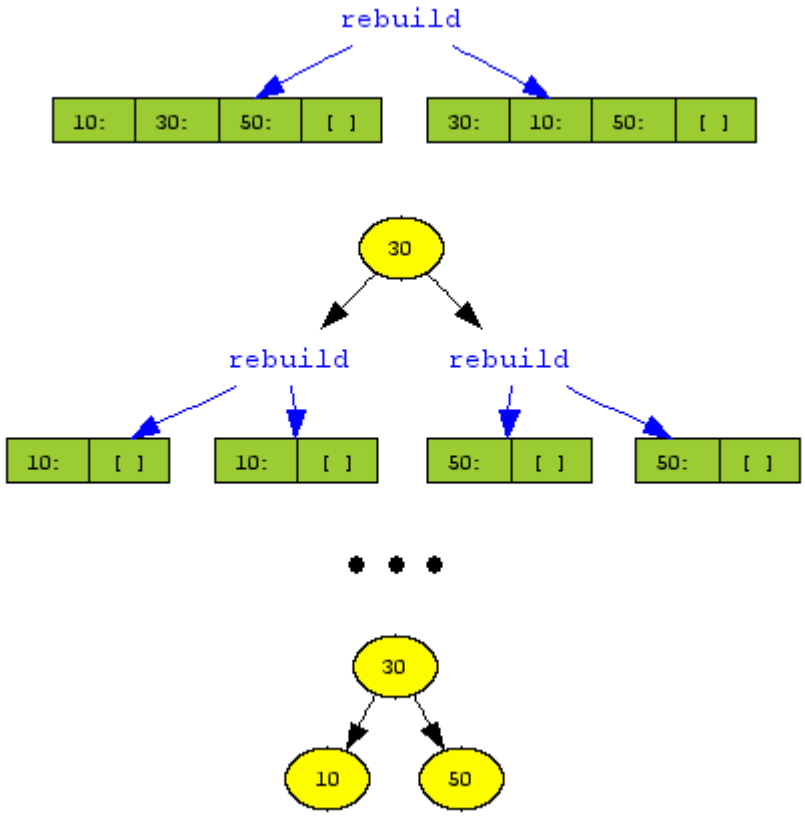


Finalmente los recorridos del último subárbol son los siguientes

preorden: **50** - 55
inorden: **50** - 55

obteniéndose:

subárbol izquierdo : vacío
subárbol derecho : 55



Ejemplo de reforestación

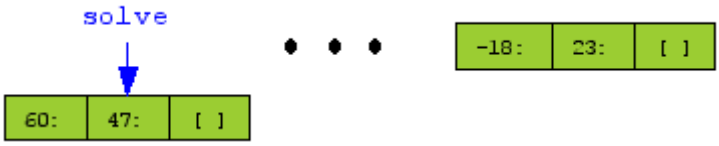
32. solvePrima.cdd

Descripción del problema: Sean "a" y "b" enteros y sea "d = mcd(a, b)". Entonces existen enteros "m" y "n" tales que "m a + n b = d" y queremos encontrarlos.

solve (a, b) = (m, n) tal que m a + n b = d = mcd (a, b)

Descripción del algoritmo: El siguiente programa lo demuestra (y calcula esos enteros) por inducción-recursión:

```
solve (a, 0) = (1, 0)     -- m a + n 0 = 1 a + 0 b = a = d
solve (a, b) = (m, n) where
        (c, r ) = (a `div` b, a `mod` b)  -- = m (b c + r) + n b
        (m', n') = solve (b, r)           -- = (m c + n) b + m r
        (m, n ) = (n', m' - m * c)        -- = ( m' ) b + n'r
```



Ejemplo de la función solve

13 Bibliografía

[Ben01] Mordachai Ben-Ari. La visualización de software en la teoría y en la práctica. Artículo citado en [Nov01]

[Enc00] Alberto de la Encina Vara. Visualización de la evaluación de programas funcionales. Apuntes para una conferencia dada en la Universidad Rey Juan Carlos de Madrid. 2000

[Fie88] Anthony J. Field, Peter G. Harrison Functional Programming 1988 Addison - Wesley

[Gam95] E. Gamma et al. Patrones de diseño. Ed. Addison-Wesley

[Gho01] GHood. A graphical visualization and animation of Haskell objects observation. http://www.informatik.uni-bonn.de/~ralf/hw2001/6.pdf. [Consulta: 5 abril 06]

[Gra] Graphviz – Graph visualization software. http://www.graphviz.org/ [Consulta: 30 marzo 2006]

[Hat] Hat publications. http://www.haskell.org/hat/publications.html. [Consulta: 3 abril 2006]

[Hoo] Hood: Haskell Object Observation Debbuger. http://www.haskell.org/hood/index.htm. [Consulta: 3 abril 2006]

[Jav] Java.net: The source for Java technology collaboration. JavaCC Project. https://javacc.dev.java.net/ [Consulta: 21 noviembre 2005]

[JVI] J. A. Velázquez Iturbide. Asignatura de Doctorado: Interacción visual persona-computador. Parte sobre Visualización del software. http://neumann.dma.fi.upm.es/mabellanas/urjc/uno.htm. [Consulta: 31 marzo 2006]

[Nov01] Novática, revista de la Asociación de Técnicos de Informática. Marzo – Abril de 2001, num. 150

[Vit] Vital homepage. http://www.cs.kent.ac.uk/projects/vital/index.html [Consulta: 14 noviembre 2005]

[Win] The Functional Programming Environment WinHipe. http://vido.escet.urjc.es/winhipe/ [Consulta: 17 noviembre 2005]

14 Apéndices

13.1 Intérprete Haskell

A continuación mostramos un pequeño intérprete realizado en Haskell que sirvió como punto de referencia a lo largo del proyecto.

```
type Ident     = String

data Expresion = Var    Ident
               | Entero Int
               | Logico Bool
               | Lambda [Ident] Expresion
               | Aplic  Expresion [Expresion]
               | Primit Ident
               | Where  Expresion [(Ident, Expresion)]
               | Condic Expresion Expresion Expresion
               | Clausura Expresion Entorno

               deriving Show

type Entorno   = [(Ident, Expresion)]

eval  :: Expresion -> Entorno -> Expresion
eval  e@(Entero z) _   = e
eval  e@(Logico p) _   = e
eval  e@(Var   id) env = case lookup id env of
                          Just expr -> eval expr env
                          Nothing   -> error (id ++ " no definido")
eval  e@(Aplic fun exprs) env  =apply(eval fun env)[eval e env | e <-exprs]
eval  e@(Primit id) _                  = e
eval  e@(Lambda parsFormales expr) env = Clausura e env
eval  e@(Clausura cl env) env'         = e
eval  e@(Where expr defs) env          = eval expr (defs ++ env)
eval  e@(Condic cond expr1 expr2) env  = case eval cond env of
                                          Logico True  -> eval expr1 env
                                          Logico False -> eval expr2 env


eval  e@(Lambda ident expr) env        = Clausura e env
eval  e@(Clausura expr env) env'       = e

apply (Primit op) argumentos = funDe op argumentos
apply (Clausura (Lambda ids e) env) es  = eval e $ (zip ids es) ++ env


-------------------------------------------------------
-- Apéndice / preludio: operaciones primitivas


funDe "+"  [Entero x, Entero y]  = Entero (x+y)
funDe "+"  [_ , _]  = error $ "'+' aplicada a dos pars. NO enteros"
funDe "+"  _  = error $ "'+' con inexacto núm. de argumentos"
funDe "-"  [Entero x, Entero y]  = Entero (x-y)
funDe "-"  [_ , _ ] = error $ "'-' aplicada a dos pars. NO enteros"
funDe "-"  _ = error $ "'-' con inexacto núm. de argumentos"
funDe "*"  [Entero x, Entero y]  = Entero (x*y)
funDe "*"  [_ , _ ]= error $ "'*' aplicada a dos pars. NO enteros"
funDe "*"  _               = error $ "'*' con inexacto núm. de argumentos"
funDe "=0" [Entero z]      = Logico (z==0)
funDe "=0" _               = error $ "'=0' aplicada erróneamente"

-------------------------------------------------------
```