Edición interactiva en entornos incrementales

Miguel Angel Alonso Pardo

September 12, 1994

A Adriana y a mi familia

Agradecimientos

Deseo expresar mi gratitud a todos aquellos que han colaborado de manera directa e indirecta, en la realización de esta tesina/proyecto de fin de licenciatura.

Entre todas esas personas destaca, como no, el Dr. Manuel Vilares, sufrido director de la tesina, siempre dispuesto a echar una mano, o las dos, sin cuyos consejos, ayuda, comprensión, sugerencias e imposiciones este trabajo nunca hubiera sido lo que es.

Importante también ha sido la ayuda prestada por Víctor Gulías, compañero de fatigas en la a veces desesperante, a veces gratificante tarea de desentrañar los secretos de gran parte del software y hardware empleado.

Inestimable ha sido la ayuda y el apoyo prestado por Jorge Graña, especialmente en las tareas relacionadas con la integración del analizador léxico, en las cuales también ha colaborado el Dr. Alberto Valderruten.

Debo agradecer también la colaboración de los profesores de la facultad, especialmente los integrantes del LFCIA[1] del Departamento de Computación por estar dispuestos a ayudar en la medida de lo posible siempre que se lo he solicitado.

Debo felicitar a Adriana Dapena y Celia González-Elipe por su labor de corrección sintáctica y léxica de los diferentes borradores y del texto final.

Muy especialmente, agradecer a Adriana su apoyo y comprensión en los momentos difíciles, en los que me alentaba a proseguir. Sin ella, nada sería igual.

[1] **L**aboratorio de **F**undamentos de la **C**omputación e **I**nteligencia **A**rtificial.

Contenido

Lista de figuras

Capítulo 1

Introducción

Como se indica en el título, en esta tesina se estudian diferentes estrategias de interfaz que se pueden utilizar para conseguir una integración eficiente y a la vez elegante de un entorno de generación de analizadores no deterministas incrementales. Conjuntamente se va mostrando el proceso de construcción de una implementación real en la que se integran los siguientes elementos:

- Un analizador sintáctico no determinista incremental para gramáticas de contexto libre.
- Un analizador léxico no determinista[1].
- El componente de interfaz gráfica que proporciona un medio agradable y eficiente de interacción con el usuario.

A continuación describimos muy sucintamente cada uno de los distintos componentes.

El analizador sintáctico se ha construido por medio de la herramienta ICE[2] desarrollada por el Vilares [Vilares 92]. Brevemente, diremos que ICE es capaz, a partir de la definición de una gramática cuyas reglas se expresan en el mismo formalismo que el utilizado por YACC, de generar un analizador sintáctico no determinista e incremental para dicha gramática.

El analizador léxico actual se ha desarrollado utilizando FLEX [Paxon 94]. Brevemente, diremos que este lexical no determinista es capaz de realizar todos los posibles análisis morfológicos de cada palabra de un texto en español partiendo de su lexema y estudiando sus morfemas. De ello se deriva un reducido tamaño y una alta eficiencia, ya que no es preciso almacenar todas las formas de las palabras del lenguaje, sino sólo su lexema y las normas generales que indican cómo se añaden los morfemas.

La interfaz gráfica se ha construido utilizando AÏDA [ILOG 92c], una herramienta para la construcción de interfaces gráficas construida sobre la base de LE-LISP [INRIA 91].

El presente trabajo se centra en la integración de los tres componentes y en la construcción del componente de interfaz. Por ello no se va a realizar un estudio detallado sobre el proceso de construcción de los analizadores sintáctico y léxico. El lector interesado

[1] La necesidad del no determinismo a nivel léxico viene dada por la finalidad de la implementación que se muestra en este proyecto: la construcción de un entorno integrado de procesamiento de lenguaje natural, concretamente aplicado a la lengua castellana.

[2] **I**ncremental **C**ontext-Free **E**nvironment.

puede recurrir a los trabajos publicados sobre ellos. Sin embargo, es interesante motivar el uso de analizadores no determistas e incrementales, que constituyen la razón de ser y el contexto en el cual se desenvuelve esta tesina.

1.1 El análisis incremental no determinista

Dentro del mundo de la informática, el desarrollo de técnicas para el diseño de analizadores sintácticos siempre ha tenido especial relevancia, principalmente debido a su estrecha relación con el mundo de los compiladores. Desde hace años, se han estudiado en profundidad una serie de técnicas que permiten realizar de un modo eficiente el análisis sintáctico o *parser* de un texto fuente de un lenguaje. Entre estas técnicas podemos citar entre las más conocidas la LL(k) de carácter descendente y las distintas variantes de la técnica LR(k) de carácter ascendente. Las técnicas ascendentes se han mostrado como las más eficientes en el tratamiento de la entrada, y de entre éstas las LR(k) brillan con luz propia. De hecho YACC [Johnson 75, Mason y Brown 90, SunSoft 93, Aho et al. 90, Kerninghan y Pike 83], el conocido programa de generación de analizadores sintácticos ampliamente usado en el mundo Unix (generalmente en compañía del generador de analizadores léxicos conocido por LEX [Mason y Brown 90, SunSoft 93, Aho et al. 90]) implementa un algoritmo LALR, extrapolado directamente a partir de los LR(k).

Sin embargo, los principales desarrollos, incluido YACC, presentan dos características que limitan su aplicación: su caracter determinístico y no incremental.

1.1.1 El análisis incremental

Tras analizar la salida generada por el compilador (que puede estar constituida por un conjunto de listados con los errores encontrados y sus referencias al texto fuente), si efectivamente éste ha detectado errores, será necesario repetir el ciclo, lo que conllevará que el texto fuente sea reanalizado completamente, aunque el error tan sólo afecte a una pequeña porción del programa. Ciertos compiladores no proporcionan un listado de todos los errores encontrados sino que paran el proceso de compilación al encontrar el primer error. El usuario debe entonces modificar el texto y recompilar el programa. En este punto no nos interesa si el compilador es llamado desde la línea de comandos o si por el contrario dispone de un entorno de programación que permite ralizar la compilación directamente desde un editor. Lo que realmente interesa resaltar aquí es que cada vez que se invoca al compilador, *todo* el texto fuente es reanalizado completamente.

Inmediatamente se puede pensar que reconstruir totalmente el árbol de análisis sintáctico constituye un derroche cuando la corrección del error tan sólo provocará la modificación de una rama de dicho árbol. De acuerdo con esto, lo ideal sería que tan sólo se reconstruyesen (o mejor dicho, se *reanalizasen*) aquellas ramas afectadas por el error. Sin embargo, para conseguir esto que aparentemente es tan sencillo se deben dar una serie de condiciones como son:

- El analizador sintáctico debe efectivamente construir una representación completa del árbol de análisis sintáctico, que debe estar disponible para el siguiente análisis.
- El analizador sintáctico debe conocer exactamente qué componentes léxicos han sido modificados por el usuario desde el último análisis.

- Debe de existir un entorno de compilación que mantenga el texto, el árbol de análisis sintáctico y las relaciones existentes entre ambos. Esto es, un editor interactivo.

El cumplimiento de estas condiciones implica una modificación sustancial del análisis sintáctico clásico, ya que:

- Los analizadores sintácticos más comúnmente usados en la actualidad, no mantienen una representación completa de las estructuras de cálculo utilizadas en el análisis sintáctico, sino que suelen utilizar una *pila* o *stack* en la que se van almacenando valores que representan el avance del proceso de análisis en un momento dado. El movimiento entre estados del autómata asociado al analizador provoca la localización de nuevos elementos, o su eliminación, de la pila. Generalmente la realización de desplazamientos conlleva la introducción de más elementos en la pila mientras que las reducciones implican la eliminación de la pila de un cierto número n de elementos a partir del tope. Dicho número n suele estar relacionado con la longitud de la parte derecha de la regla. De este modo se consigue un reconocedor muy eficiente tanto en tamaño como en velocidad, pero al finalizar el proceso de análisis se carece de una representación completa del árbol.

- Para que en un análisis incremental de un texto previamente analizado el analizador pueda saber qué parte del árbol debe ser reconstruida, éste debe poseer algún conocimiento sobre las modificaciones que se han realizado sobre el texto fuente y cómo han afectado a los componentes léxicos. Para conseguirlo es necesario integrar el analizador léxico con el texto de modo que el editor sea capaz de establecer las conexiones componente léxico-texto y pueda guiar al usuario en las operaciones de modificación, al mismo tiempo que debe ser capaz de indicar al analizador sintáctico qué porciones del análisis anterior han de ser revisadas. Es en este trabajo de integración y de construcción del entorno común parser-lexical-usuario en lo que se centra la mayor parte de este proyecto

En el procesamiento del lenguaje natural el uso de analizadores incrementales presenta más ventajas incluso que en el campo de los compiladores de lenguajes de programación, ya que permiten que ante una entrada errónea (una falta de ortografía, un error al realizar el OCR[3] de un documento digitalizado mediante un escáner, etc.) sólo se tenga que reanalizar como mucho la frase en la cual está contenido el error. En este contexto, sería prohibitivo que para subsanar un error se tuviese que realizar un nuevo análisis completo de todo el texto de entrada.

1.1.2 El análisis no determinista

La utilización prácticade técnicas no deterministas en la construcción de compiladores para lenguajes de programación es escasa. Las técnicas clásicas como LL(k) y LR(k) son deterministas. Esto quiere decir que para una entrada dada (un programa) sólo se obtendrá un árbol de análisis sintáctico. Como no todas las gramáticas de contexto libre pueden ser transformadas de modo tal que sean reconocidas por un autómata determinista, estas técnicas no reconocen todas las gramáticas de contexto libre, aunque sí las utilizadas

[3] **O**ptical **C**haracter **R**ecognition.

en todos los lenguajes de programación. Algunas herramientas basadas en técnicas determinísticas como YACC permiten que la gramática sea ambigüa, pero a la hora de realizar el análisis, cuando se presentan *conflictos* siguen normas bien conocidas para resolverlos. Por ejemplo, cuando YACC detecta un conflicto de desplazamiento/reducción, opta siempre por realizar el desplazamiento, lo cual le lleva a comportarse bien ante casos como el del *else ambiguo*.

En el procesamiento del lenguaje natural el uso del no determinismo es obvio ya que los idiomas como el castellano presentan por su propia naturaleza cierto número de ambigüedades a nivel léxico y sintáctico.

Para que un analizador sintáctico sea capaz de tratar con el no determinismo debe ser capaz de tratar con más de un árbol sintáctico. Para no perder eficiencia se debe de buscar un método que permita compartir la mayor cantidad posible de información, ya que mantener varias representaciones completas del mismo árbol cuando tan sólo se diferencien en una serie de ramas (aquellas que corresponden a los distintos caminos a tomar cuando tenemos una ambigüedad) representaría un consumo de memoria excesivamente alto.

ICE es capaz de generar eficientes analizadores no deterministas incrementales, como se puede observar en las resultados empíricos mostrados en [Vilares 93, Vilares y Dion 94]. ICE permite incorporar la incrementalidad y el no determinismo al proceso del análisis sintáctico sin que ello suponga un coste elevado en cuanto a eficiencia, e incrementa en gran medida la flexibilidad en lo concerniente al diseño de gramáticas.

1.2 Antecedentes

En los entornos tradicionales de carácter determinista y con ausencia de incrementalidad se han realizado varios trabajos que, al menos en apariencia, poseen interfaces integradas con un analizador sintáctico. De tal modo, existen en el momento actual herramientas, que a veces reciben nombres tales como *entornos de desarrollo integrados*, que poseen un editor integrado en cierta medida con la sintaxis que es capaz de mostrar al programador una representación más visual del código fuente mediante la utilización de distintos recursos gráficos (principalmente una cierta gama de distintas fuentes y colores) para indicar palabras clave, tipos, variables, constantes, etc. En estos desarrollos, aunque ya existe un cierto grado de integración entre los componentes, ésta es débil y está más orientada hacia una simple presentación más *visual* e intuitiva del código fuente que a una real interacción interfaz-parser-lexical.

En el campo del análisis incremental existe un interesante desarrollo llamado ASF+SDF [Centaur 92a, Centaur 92b, Centaur 92c] que combina un formalismo de especificación algebraica[4] con un formalismo de definición de sintaxis[5], integrados ambos en un editor dirigido por la sintaxis denominado GSE[6] [Centaur 92b]. SDF permite especificar cualquier sintaxis de contexto libre mediante un formalismo propio. Se puede validar el analizador para dicha gramática utilizando un texto en un editor de tipo GSE. En dicho editor, parte del texto está resaltado en lo que los autores denominan *focus*. Sobre el focus se pueden realizar acciones de *zoom* de tal modo que en vez de texto pasa a representar un nodo interior del árbol sintáctico. La característica más destacable es que se puede

[4] ASF, de **A**lgebraic **S**pecification **F**ormalism.
[5] SDF, de **S**yntax **D**efinition **F**ormalism.
[6] **G**eneric **S**yntax-directed **E**ditor.

modificar la especificación de la gramática y automáticamente se realizará un análisis incremental en el texto de prueba trasladando el focus a aquella porción de texto que engloba los posibles errores sintácticos surgidos como resultado de la transformación de la gramática. SDF utiliza un autómata LR(0) extendido cuya construcción es incremental a partir de la nueva gramática modificada y el autómata correspondiente a la versión original de la misma. El análisis sintáctico en sí mismo no es incremental. En cuanto al no determinismo, SDF dispone de mecanismos de asignación de prioridades a las reglas que se utilizan en el proceso de desambigüación.

Capítulo 2

Asociación de los componentes léxico con el texto

En este capítulo se va a discutir un conjunto de estrategias entre las cuales se puede optar para construir un *editor de componentes léxicos*. Este editor constituye uno de los principales componentes de cualquier aplicación que pretenda conseguir un interfaz eficiente y consistente en entornos de análisis incremental, puesto que será el encargado de realizar la mayor parte de las tareas involucradas en la interacción usuario-analizador. Su correcto diseño es vital para conseguir una herramienta operativa.

2.1 Introducción

La tarea de construir el editor de componentes léxicos puede subdividirse, en nuestro caso, en el siguiente conjunto de actividades:

1. Diseñar las estructuras de datos subyacentes que den soporte adecuado a las operaciones de edición.

2. Implementar dichas estructuras utilizando las construcciones del LE-LISP y AÏDA que permitan obtener una mayor eficacia y una mayor economía en recursos computacionales.

3. Construir el caparazón externo, la visión que se ofrecerá al exterior, ocultando las complejidades internas. Constituye lo que se ha dado en llamar la interfaz de usuario.

En este capítulo se tratan los puntos 1 y 2. El tercer punto se tratará en el capítulo 3. Se estudian diferentes estrategias de diseño que pueden seguirse para asociar de modo eficiente los componentes léxicos del análisis con su representación textual. Para cada una de las estrategias se muestra esquemáticamente una posible implementación. Concretamente se van a estudiar en detalle los beneficios e inconvenientes de las tres siguientes:

- Estrategia de *asociación directa*, consistente en crear una estructura de representación independiente para cada uno de los componentes léxicos.

- Estrategia de *asociación indirecta*, según la cual se construye una estructura de datos global para todos los componentes léxicos generados por el texto. Se pueden distinguir al menos dos variantes:
  - *asociación indirecta por posición*
  - *asociación indirecta por desplazamiento*
- Estrategia de *asociación multinivel*, que consiste en establecer una estructura global mediante la cual se accede a otras estructuras que representan la asociación entre el texto y los componentes léxicos para zonas concretas del archivo de texto.

2.2 Asociación directa

En esta primera estrategia se realiza de modo simultáneo la tarea de reconocimiento de cada componente léxico y la creación de una estructura de datos que represente el enlace entre dicho componente léxico y los caracteres correspondientes a su representación textual. En dicha estructura se almacenará tanto la información de análisis del componente léxico que sea relevante para el usuario como la información de su representación gráfica en la pantalla. Con ello conseguiremos:

- Mantener unificada toda la información relativa a un componente léxico, tanto la proporcianada por el proceso de análisis como la concerniente a la imagen gráfica con que se muestra al usuario.
- Disponer de un método directo para implementar la interacción con el usuario, ya que cuando éste señala la imagen gráfica del componente léxico, el acceso a sus datos de análisis es inmedianto.
- Aislar los componentes léxicos, ya que las operaciones realizadas sobre uno de ellos no afectan a los demás.

2.2.1 Tratamiento de los componentes léxicos

Para construir la estructura de representación de cada componente léxico se utiliza un método de construcción directa que se adapta muy bien a la filosofía de la estrategia, puesto que permite obtener de modo inmediato toda la información requerida para cada componente léxico. Para ello se deben seguir los siguientes pasos:

- Si se está realizando el análisis inicial del texto, por cada componente léxico reconocido en el análisis léxico se realizan las acciones siguientes:
  1. Crear la estructura de datos correspondiente a dicho componente léxico, rellenando las partes correspondientes a la información de análisis.
  2. Crear una instancia del *widget*[1] que se corresponde con la representación gráfica del componente léxico e incorporarlo a su estructura de datos.

[1] Los widgets son los bloques gráficos básicos estándar mediante los cuales se crea y unifica el aspecto de una interfaz de usuario

3. Mostrar en pantalla el widget del componente léxico. Para ello primero deberemos eliminar la porción del texto que se corresponde con la cadena reconocida por el lexical para posteriormente ubicar en su sitio la nueva imagen.

- Si se está realizando un procesamiento incremental de la entrada, tan sólo se deberá actuar sobre las partes del editor que no se correspondan con las representaciones de los componentes léxicos de la parte estable, ya que éstos últimos no se verán afectados por el nuevo análisis. Con la parte no estable se procederá como sigue:

  1. Eliminamos de pantalla los widgets de la representación gráfica de dichos componentes léxicos mostrando en su lugar simples cadenas de caracteres. Con esto conseguiremos restaurar a su estado original la parte del editor de componentes léxicos que participa en el nuevo análisis, con lo cual es factible considerar, en lo concerniente a los aspectos de representación gráfica, dicha porción como un editor de componentes léxicos independiente que contiene un texto sin analizar.
  2. Según va avanzando el proceso de análisis, se deben ir modificando las estructuras de datos de los componentes léxicos correspondientes.
  3. Con respecto a los elementos gráficos, se procede como en los pasos 2 y 3 del análisis inicial.

La estructura de datos que da soporte a la asociación deberá contener, como mínimo los siguientes datos:

- El identificador del componente léxico, que en ICE es un número entero que los identifica unívocamente.
- La información de análisis del componente léxico a la que pueden acceder los usuarios.
- El widget que contiene la representación gráfica del componente léxico

2.2.2 Conclusiones

En principio, la estrategia de asociación directa parece constituir una elección acertada ya que proporciona, mediante una perspectiva sencilla, una construcción de las estructuras de representación de los componentes léxicos que cumple los requisitos y realiza las funciones especificadas en los requerimientos del editor de componentes léxicos[2]. Sin embargo, si tomamos en cuenta las consideraciones de rendimiento observamos que esta estrategia presenta serios inconvenientes. La razón estriba en la carga computacional asociada a la gestión de los TGR, ya que es necesario construir un TGR por cada componente léxico presente en el texto, lo que provoca un elevado consumo de memoria y de tiempo de procesamiento para manejarlos en pantalla. Como consecuencia, no se cumple el requisito de que el tiempo dedicado al procesamiento de las funciones propias del editor no debe suponer una sobrecarga significativa del proceso de análisis sintáctico.

Por tanto, esta estrategia tiene como puntos a favor su sencillez y elegancia, pero su utilidad se ve seriamente limitada debido a sus problemas de rendimiento, lo que provoca que no sea aplicable en la práctica.

[2] Ver sección 3.2.1 en la página 34.

2.3 Asociación indirecta

En la estrategia de asociación indirecta se realiza simultáneamente el proceso de reconocimiento de los componentes léxicos y la creación de una estructura de datos global que almacena la información relativa al análisis de cada uno de ellos. Como consecuencia, se debe definir un mecanismo indirecto de acceso a esa información a partir del texto, puesto que ya no es posible crear una aplicación propia para cada uno de los componentes léxicos. Con ello conseguiremos evitar la sobrecarga computacional y el elevado consumo de recursos que supone la utilización de una estrategia de asociación directa.

El núcleo de esta estrategia lo constituye precisamente dicha estructura global, a la que podemos nombrar TTLT[3], por lo que la mayor parte de la discusión de la presente estrategia se centra en estudiar la mejor manera de lograr una implementación de la TTLT que permita un enlace eficiente con el texto contenido en el editor.

A diferencia de lo que ocurría en el caso de la asociación directa, ahora no se realiza una transformación del texto del editor en imágenes gráficas cuando se analiza el texto, sino que se mantiene el texto original en todo momento. El aspecto externo del editor de componentes léxicos permanece invariable, puesto que precisamente con la utilización de un widget para cada componente léxico en la estrategia de asociación directa se intentaba conseguir un aspecto homogéneo de texto simple en todo el editor a la vez que se proporcionaba una estructura subyacente no visible al usuario que facilitaba la interacción de éste con los componentes léxicos. En definitiva, en la estrategia de asociación indirecta se trata de mantener la interfaz externa con el usuario a la vez que se proporciona un mecanismo interno más eficiente en lo que respecta a la interacción del texto con los componentes léxicos.

El proceso a seguir cuando se desea acceder al componente léxico que se corresponde con una determinada porción de texto es el siguiente:

1. A partir de las coordenadas, absolutas o relativas, del carácter situado en el punto en el que el usuario hizo click con el ratón, se calcula una clave de acceso al elemento de la TTLT que corresponde al componente léxico cuyo texto está situado en dicha posición en el editor.

2. Mediante la clave calculada en el paso anterior se accede a la informacón de análisis almacenada en la TTLT para el componente léxico.

El tiempo medio transcurrido desde que se solicita acceso a un componente léxico hasta que se accede realmente ha de ser igual para todos los componentes léxicos, independientemente de la posición en el editor en la que se encuentre el texto correspondiente.

2.3.1 Construcción de la TTLT

Para que esta estrategia sea aplicable debe diseñarse de modo que su consumo de memoria sea reducido. El tiempo de procesamiento consumido en el mantenimiento de la representación de las operaciones realizadas sobre el texto debe mantenerse dentro de límites aceptables. Debido al elevado número de componentes léxicos que potencialmente

[3] **T**oken-**T**ext **L**ink **T**able, Tabla de enlace componente léxico-texto.

contendrá cada texto que será sometido a análisis, del orden de decenas de miles, se debe de buscar una representación para la TTLT que cumpla los siguientes requisitos:

- Sea capaz de almacenar en memoria un elevado número de elementos, cada uno de los cuales se encargará del almacenamiento de la información relativa a un componente léxico.
- Tenga un carácter dinámico, es decir, no debe establecerse una limitación sobre el número máximo y mínimo de elementos que limiten la flexibilidad de la herramienta.
- Debido a las características del análisis incremental, debe también optimizar la reutilización de la memoria liberada por los elementos que se correspondan con componentes léxicos eliminados.
- Proporcione un mecanismo de acceso eficiente a cada elemento mediante algún tipo de clave.

En LE-LISP disponemos de varias alternativas para implementar la TTLT:

- Las listas, que por su propia naturaleza son extensibles dinámicamente. Utilizando las funciones de manejo que modifican físicamente sus argumentos se mejora el uso que hacen de la memoria, evitando problemas con el *garbage collector*[4]. Su principal inconveniente radica en el tiempo de acceso a los elementos: para acceder a un elemento dado de una lista hay que recorrer la lista desde el principio hasta encontrarlo. Esta forma de acceso secuencial implica además que no todos los componentes léxicos tiene un tiempo medio de acceso igual, sino que aquellos que estén situados al principio de la lista dispondrán de un acceso extremadamente rápido mientras que aquellos situados cerca de la cola sufrirán un elevado retardo.
- Los vectores, que presentan como ventaja respecto a las listas la posibilidad de un acceso directo a cada elemento. Sin embargo, su limitación radica en que su tamaño no puede variar dinámicamente, por lo que no son aplicables en este caso. Una limitació añadida reside en el hecho de que las implementaciones actuales de los vectores en LE-LISP tiene fijado un límite máximo de 32767.
- Las tablas hash tienen tamaño dinámico, manejan eficientemente el espacio de memoria y el tiempo de acceso a cada elemento es constante independientemente de su posición. Como ventaje añadida está que el formato de la clave de acceso puede ser fijado libremente por el programador, lo cual aporta un grado de flexibilidad del que no se dispone en el caso de las listas y los vectores.

La mejor alternativa consiste en utilizar tablas de hash como estructura de almacenamiento global. Una vez que ha sido resuelto el problema de la TTLT surge otro relativo a la forma que van a tener sus elementos. En efecto, mediante las tablas hash se consigue una TTLT flexible y eficiente en el manejo de la memoria y del tiempo de acceso a sus elementos, pero su eficiencia se verá seriamente limitada si los elementos que en ella se van a almacenar se construyen de tal modo que consuman elevadas cantidades de memoria y retarden el acceso a la información que contienen.

[4] Recolector de basura.

2.3.2 Variantes de la estrategia de asociación indirecta

Se pueden distinguir una serie de variantes de la estrategia de asociación indirecta que se diferencian fundamentalmente en los siguientes aspectos:

- La forma de acceso a la TTLT.
- La forma en que se calculan las claves correspondientes a la entrada de un componente léxico.
- La información contenida en los TTR.

El problema de resolver el enlace entre el texto y los componentes léxicos se puede ver como el problema de buscar la mejor manera de indexar ese texto de tal modo que se puedan construir claves adecuadas para el acceso a la TTLT. No se trata de un problema banal, ya que un método de construcción de las claves debe de satisfacer varios requisitos para que sea considerada válido, entre los cuales destacan:

- No deben existir conflictos, esto es, no se debe dar el caso de que las claves de dos componentes léxicos distintos sean iguales, puesto que en tal caso un TTR estaría asociado a dos componentes léxicos distintos, violando las características de diseño de la TTLT. Cierto es que existen técnicas para resolver los conflictos surgidos por claves que apuntan al mismo elemento en tablas hash, tales como el encadenamiento de elementos o la creación de áreas de overflow. Sin embargo, esto supone aumentar la carga de procesamiento, por lo que es deseable diseñar las claves y sus algoritmos de transformación de tal modo que no surjan conflictos.
- Las claves se deben poder construir a partir de componentes obtenibles fácilmente del texto, evitando tener que utilizar transformaciones complejas que penalicen el rendimiento.

2.4 Asociación indirecta por posición

Una forma de indexar la TTLT que cumple las condiciones mencionadas en la sección anterior consiste en utilizar la posición del editor en la cual aparece el primer carácter de cada componente léxico como clave de acceso. El proceso a seguir se detalla a continuación:

1. Transformar la coordenada horizontal en una cadena de caracteres.
2. Concatenar la cadena de caracteres obtenida en el paso 1 con un caráter separador, por ejemplo una coma.
3. Concatenar la cadena obtenida en el paso 2 con el resultado de convertir en string la posición vertical del texto del componente léxico.

La utilización de un separador (en este caso una coma, aunque podría ser cualquier carácter no numérico) se debe a la necesidad de evitar la duplicidad de claves, ya que de no usar separador podría darse el caso de componentes léxicos distintos que, debido al proceso de concatenamiento, produjesen la misma clave. Por ejemplo, dos componentes léxicos

cuyos textos comenzasen en las posiciones (10,15) y en la (101,5) respectivamenente, tendrían ambos asociada la cadena "1015". Mediante la introducción de la coma entre las coordenadas evitamos tales problemas. En el ejemplo, se obtendrían las cadenas "10,15" y "101,5" como claves a través de las cuales acceder a la TTLT.

2.4.1 Acceso a los componentes léxicos

Cuando el usuario haga click con el ratón sobre el texto con el fin de indicar que desea realizar una operación de modificación sobre el componente léxico asociado al texto que está señalando, se utilizan las coordenadas del texto para construir la clave con la cual acceder a la TTLT y recuperar el TTR asociado al componente léxico indicado. Sin embargo, aun quedan dos cuestiones importantes por determinar:

- El mecanismo mediante el cual se determina la posición inicial del texto correspondiente a un componente léxico.
- La información que se deberá almacenar en los TTR para poder soportar la operaciones de edición.

Respecto a la primera de las cuestiones planteadas, existen en principio dos métodos mediante los cuales proceder a su resolución:

- Crear durante el proceso de análisis las claves correspondientes a todas las posiciones de los caracteres que forman el texto de cada componente léxico y utilizarlas como entradas en la TTLT que apunten al TTR del componente léxico correspondiente. De este modo, cuando el usuario pulse el botón del ratón para indicar una operación, se puede utilizar la fila y la columna del carácter situado justo bajo el cursor para construir la clave a través de la cual acceder a la TTLT.
- Crear una entrada en la TTLT utilizando como clave de acceso las coordenadas del primer carácter del texto asociado al componente léxico. Para el acceso se debe construir un conjunto de funciones que busquen el carácter más cercano que tenga una entrada en la tabla.

El primer método presenta, en principio, los siguientes inconvenientes:

- Redundancia de información, ya que cada componente léxico tendrá asociadas tantas entradas de la TTLT como caracteres posea el texto que lo representa.
- Actualizaciones múltiples, como consecuencia del punto anterior, ya que durante el proceso de análisis incremental será necesario modificar todos los TTR de un componente léxico con el fin de mantener la consistencia de la información almacenada en ellos.
- Gran tamaño de la TTLT, ya que tendrá una entrada por cada carácter que aparezca en el texto.

Una solución inmediata consiste en hacer que las entradas de la TTLT almacenen un puntero a los TTR, como se suele hacer en lenguajes de programación imperativos clásicos como el C o el Pascal. En LISP, al almacenar elementos en una tabla hash lo que realmente se guardan son punteros al objeto, por lo que si no median operaciones

explícitas de creación, dos operaciones que almacenen la misma variable en entradas distintas tan solo crean dos enlaces a la misma variable (zona de memoria). Esto tiene efectos perniciosos cuando queremos utilizar, por ejemplo dentro de un bucle, una misma variable para almacenar varios elementos sucesivamente en una tabla hash, ya que en realidad se están almacenando punteros a misma variable con lo cual todas las entradas tendrán asociado el último valor asignado a la variable. Hay que tener mucho cuidado con este tipo de situaciones, por otro lado bastante frecuentes en LISP, aunque en esta ocasión puedan resultar ventajosas.

El tener que crear una entrada en la TTLT por cada carácter presente en el texto hace que la TTLT alcance tamaños prohibitivos para textos medianamente grandes. Esto se convierte en el mayor incoveniente de esta forma de acceso a la TTLT.

El segundo método tiene como ventaja unos menores requerimientos de memoria, ya que para cada componente léxico sólo se almacena una entrada en la TTLT. Como contrapartida se encuentra la mayor craga de procesamiento asociada, ya que para facilitar la interacción con el usuario se permite que éste apunte sobre cualquier carácter del texto y no sólo sobre aquellos mediante los cuales se realiza la indexación de la tabla. En este método se necesita definir un conjunto de funciones cuya misión será buscar el componente léxico al que pertenece el carácter señalado.

2.4.2 Contrucción del editor

En base a la discusión precedente, para implementar la estrategia de asociación indirecta por posición se han tomado las siguientes opciones de construcción:

- Utilizar una tabla hash para construir la TTLT.
- Utilizar vectores para construir cada uno de los TTR.
- Almacenar en la TTLT un sólo TTR por componente léxico analizado, utilizando la posición del primer carácter su texto asociado para indexar la tabla.

2.4.3 Las funciones de búsqueda de componentes léxicos

Para conseguir un editor operativo es necesario definir un conjunto de funciones cuya misión es la de buscar el componente léxico a cuyo texto pertenece el carácter que el usuario ha indicado. Estas funciones son:

- `search-ttr-forward` que busca hacia adelante en el texto una entrada en la TTLT que se corresponda con un TTR. Esta detección es posible ya que los intentos de acceso a entradas vacías de una tabla hash producen como resultado el valor `()`.
- `search-ttr-backward` que realiza la misma función que la anterior pero mediante una búsqueda hacia atrás en el texto.
- `search-ttr-backward-and-forward` que utiliza una búsqueda combinada en ambas direcciones y puede construirse combinando las otras dos.

Cuando el usuario pulsa en una posición, se utilizan estas funciones de búsqueda para encontrar el caracter correspondiente al comienzo del componente léxico más cercano.

Hay que realizar un seguimiento del texto modificado para mantener la consistencia de las entradas en la TTLT. Se pueden considerar dos casos distintos según sea el alcance de las modificaciones realizadas:

- Modificaciones que solamente afectan a una línea.
- Modificaciones que afectan a varias líneas.

En el primer caso basta con reorganizar las entradas de la TTLT correspondiente a la línea en la cual se realizó la modificación, tarea que puede ser realizada en poco tiempo. Sin embargo, en el segundo caso, en el cual las modificaciones sobre el texto de un componente léxico se extienden más allá del límite de una línea, surgen problemas de rendimiento. Este caso se da cuando insertamos un nuevo componente léxico cuyo texto contiene saltos de línea, cuando borramos un componente léxico que se extiende más allá de una sola línea o cuando al modificar un componente léxico insertamos o borramos saltos de línea. Este tipo de operaciones obligan a recalcular todas las entradas en la TTLT para aquellos componentes léxicos situados en las líneas siguientes.

2.4.4 Puntos de sincronización

Mantener actualizada la TTLT en todo momento, esto es, cada vez que se inserta o borra un carácter, supone una carga computacional excesiva. Una solución consiste en establecer *puntos de sincronización* en los que realizar la actualización. De este modo, si s_i ha sido el último punto de sincronización alcanzado, se debe garantizar que la información contenido en la tabla es consistente con el texto almacenado en el editor cuando se alcanzó s_i. Sin embargo, entre s_i y s_{i+1} la TTLT no reflejará la situación actual del texto editado. La información de la tabla será actualizada en cuanto se alcance s_{i+1} utilizando para ello los datos *viejos* de la TTLT (los correspondientes a s_i) y cierta información recopilada durante el intervalo transcurrido entre los dos puntos de sincronismo consecutivos de forma que tan sólo se actualice aquella parte de la tabla que realmente se haya visto afectada por las modificaiones introducidas en el texto durante ese periodo.

Elección de los puntos de sincronización

La opción más viable con respecto a los puntos de sincronización consiste en utilizar para tal fin la terminación de cada operación de edición, puesto que una de las acciones a realizar al comienzo de la siguiente operacion consiste en encontrar el componente léxico al que pertenece el carácter señalado por el usuario. Para ello, es necesario que dispongamos de información actualizada sobre las asociaciones entre los propios componentes léxicos y el texto correspondiente.

Por consiguiente, se debe establecer un punto de sincronismo en los siguientes casos:

- Cuando el usuario indica el tipo de la siguiente operación a realizar (mediante la pulsación de un botón o la selección de una determinado opción de menú), puesto que ello implica la finalización de la operación anterior.
- Cuando el usuario pulsa el ratón sobre el texto, está activado alguno de los modos de edición de componentes léxicos (inserción, borrado, modificación) y la acción anterior no fue seleccionar un nuevo tipo de operación. En este caso el usuario está indicando

su deseo de realizar una nueva operación del mismo tipo que la precedente, lo cual implica la finalización de la operación anterior.

- Cuando se realiza una petición de análisis del texto si se ha realizado alguna operación de edición desde el punto de sincronismo anterior.

La información de operación

Para lograr una actualización consistente es necesario disponer de la siguiente *información de operación* referida a las modificaciones introducidas en el texto desde el último punto de sincronismo[5]:

- Coordenadas de inicio de la cadena de texto afectada.
- Coordenadas finales de la cadena de texto afectada.
- Longitud del texto afectado.
- Incremento vertical del texto, que será positivo si se han añadido líneas, negativo si el número de éstas ha disminuido o cero si se conserva el número de líneas original.
- Incremento horizontal del texto, que indica la distancia horizontal entre el primer y el último carácter del componente léxico.

Los tres últimos datos se pueden derivar de los dos primeros, pero se facilitan las cosas si se mantinene actualizados conforme se va modificando el texto. Para mantener actualizada la información de operación conforme se van insertando o borrando caracteres, es necesario capturar los eventos generados por el teclado.

Se deben definir dos funciones, `insert-character` y `delete-character`, que serán llamadas cuando se produzcan modificaciones interactivas en el editor. Cada vez que se produce una modificación interactiva en el texto almacenado en el editor se llama a la función `insert-character`. Es importante reseñar que dicha función es invocada una vez que el texto ya ha sido modificado. Por modificación interactiva se entiende aquella que ha sido realizada por el usuario haciendo uso del teclado o del ratón[6]. Por tanto, no se consideran interactivas aquellas modificaciones realizadas mediante funciones definidas por el programador.

Mediante `set-local-binding` se atrapan los eventos de teclado provocados por la pulsación de las teclas DELETE y BACKSPACE[7] sobre el texto del editor. La pulsación de dichas teclas ya no provocará el borrado de un carácter, sino que hará que se invoque a la función `delete-character`.

La función `insert-character`[8] recibirá como argumentos la instancia del editor involucrada y el comando correspondiente a la acción realizada sobre el texto. La misión de esta función será:

[5] Se debe recordar que las modificaciones sólo pueden afectar a una cadena contigua de caracteres, ya que el texto correspondiente a un componente léxico no puede estar diseminado por distintos lugares del editor y tan sólo se permite editar un componente léxico en cada operación.

[6] Se puede insertar texto mediante el ratón utilizando los mecanismo de cortar y pegar

[7] En condiciones normales ambas teclas provocan el mismo efecto sobre el texto contenido en una instancia de **`textedit`**: borran el carácter que precede a la posición del cursor

[8] Esta función se describe detalladamente en la sección 3.8.2 de la página 59.

- Comprobar que no se inserte texto fuera del componente léxico que se está editando, para lo cual se comparará la posición del caracter insertado con las coordenadas iniciales y finales de la cadena de texto afectada.
- Actualizar la información de operación, teniendo especial cuidado cuando lo que se inserta es un salto de línea, que se detectan porque en tal caso el comando pasado como argumento será `te-interactive-line-break`.
- Hacer que el carácter insertado sea mostrado en pantalla utilizando la fuente y los colores correspondientes a la operación de edición de componentes léxicos que se esté realizando.

Por su parte, la función `delete-character`[9] tan sólo recibirá como argumento la instancia del editor involucrada. La misión de está función será:

- Comprobar que no se trata de borrar un carácter situado fuera del componente léxico que se está editando, para la cual se hará uso de de las coordenadas iniciales y finales de la información de operación.
- Actualizar la operación de información, teniendo especial cuidado con el borrado de caracteres de salto de líneas. A diferencia de lo que ocurría en el caso de la inserción, no existe ninguna diferencia entre borrar un LINE-FEED o CARRIAGE-RETURN y borrar cualquier otro carácter. Una forma de resolver este problema consiste en disminuir la longitud del texto afectado en una unidad y mover el cursor a partir de las coordenadas iniciales para calcular las nuevas coordenadas finales[10]. Esto hace que el borrado de un carácter tenga asocida una mayor carga computacional que una inserción, debido a la carencia de información sobre el tipo de carácter borrado.

2.4.5 Sincronización de la TTLT

Una vez que se ha mostrado cómo mantener actualizada la información de operación se puede pasar al tema de la actualización de la TTLT en los puntos de sincronización. Como ya se ha dicho anteriormente, se pueden distinguir dos casos, basándose en si la operación de edición ha variado el número de líneas del texto asociado con un componente léxico.

En el caso de que el número de líneas permanezca constante, es suficiente con proceder a un movimiento de las entradas de los componentes léxicos en la TTLT igual al incremento en la longitud de la última línea del texto asociado al componente léxico. Esto quiere decir que si un componente léxico tenía asociado un texto de longitud n y después de editado la longitud cambia a $n + \delta_x$, donde δ_x puede ser un entero positivo (incremento de longitud) o negativo (borrado de caracteres), las entradas de los componentes léxicos cuyo texto comienza en la misma línea en la que finaliza el modificado verán deplazados su posición horizontal en δ_x unidades. Las coordenadas verticales no se verán afectadas.

Si por el contrario el número de líneas varía, se deberán desplazar todas las entradas de todos los componentes léxicos posteriores al modificado. Por tanto, si un componente

[9] Esta sección se describe con más detalle en la sección 3.8.3 de la página 60.

[10] Esta estratagema funciona bien debido a que **`textedit`** también cuenta los caracteres de ruptura de línea a la hora de realizar los movimientos con el cursor, de tal modo que si el cursor está situado en el primer carácter de una línea de n caracteres, se necesitan $n + 1$ movimientos del cursor hacia la derecha para que se sitúe en el primer carácter de la línea siguiente.

léxico tenía asociado un texto que finalizaba en la posición vertical y y ahora lo hace en la $y + \delta_y$, todas las entradas de los componentes léxicos situados desde la fila y hasta el final del texto deben desplazase δ_y unidades. Si δ_y es positivo significa que se ha producido un aumento en el número de filas y si es negativo que se ha reducido el número de filas. Una vez realizado este desplazamiento vertical, los componentes léxicos situados en la línea de terminación del texto el componente léxico modificado sufrirán un desplazamiento horizontal de acuerdo con el caso anterior.

Puesto que la TTLT había perdido su coherencia con el texto, para realizar los desplazamiento de componentes léxicos es preciso examinar todas las entradas posibles en la tabla, es decir, para cada carácter en el resto de la línea o en el resto del texto según sea el caso. Esto supone un coste muy elevado en términos de recursos computacionales, especialmente en el caso de que se haya modificado el número de líneas.

2.4.6 Conclusiones

En un texto de varios cientos de líneas y varias decenas de caracteres por línea, la reorganización de la TTLT penaliza demasiado el rendimiento puesto que se dispara el número de operaciones que se deben de realizar.

Sin embargo, en aquellos casos en los que no se permite que un componente léxico tenga asociadas varias líneas de texto[11] la estrategia de asociación indirecta por posición presenta un rendimiento aceptable, puesto que tan sólo se han de reorganizar a lo sumo los componentes léxicos de una línea.

En todo caso, el consumo de memoria es mucho menor que en el caso de la estrategia de asociación directa puesto que la relación entre los componentes léxicos y el texto se guarda en una tabla global en lugar de crear aplicaciones independiente para cada componente léxico, sin que esto afecte a la presentación del editor ni a la interacción con el usaurio.

2.5 Asociación indirecta por desplazamiento

Otra variante de la estrategia de asociación indirecta consiste en indexar la TTLT directamente por el número de componente léxico y almacenr en campos de los TTR's el desplazamiento con respecto al inicio del texto del componente léxico precedente y la longitud del texto asociado al componente léxico[12]. En el caso del primer componente léxico, puesto que no existe un componente léxico precedente, el desplazamiento se refiere al inicio del texto.

Mediante esta estrategia se obtienen las siguientes ventajas, todas ellas conducentes a la obtención de una mayor independencia entre las operaciones de edición y las operaciones de análisis:

- Cada componente léxico tiene asociado una única entrada en la TTLT.
- Cuando se modifican componentes léxicos existentes no es necesario mover los TTR's asociados de unas entradas de la tabla a otras, sino que permanecerán asociados a la misma entrada.

[11] Por ejemplo, en muchos lenguajes de programación los saltos de línea actúan como separadores entre componentes léxicos.

[12] Es preciso almacenar la longitud debido a que pueden existir caracteres separadores que no formen parte de ningún componente léxico.

- La localización de las entradas en la TTLT sólo se ve afectada cuando, como consecuencia de un análisis incremental, se necesitan crear nuevas entradas para nuevos componentes léxicos y borrar entradas para componentes léxicos que han sido desechados en el nuevo análisis.
- Eliminación de la necesidad de puntos de sincronización, como consecuencia de lo anterior.

2.5.1 Acceso a los componentes léxicos

Como siempre, cuando el usuario pulsa con el ratón sobre una posición en el texto, es necesario buscar el componente léxico al cual pertenece el carácter señalado. Para ello es preciso recorrer la TTLT desde el principio hasta encontrar un TTR tal que su texto asociado incluya dicho carácter, ya que no se dispone de una asociación por posición sino por desplazamiento desde el origen. Al tratarse de una búsqueda lineal el tiempo medio de acceso al TTR buscado será igual a $n/2$, donde n se refiere al número de componentes léxicos reconocidos en el proceso de análisis[13].

Se puede considerar la opción de almacenar en cada TTR las coordenadas de comienzo del texto de cada componente léxico. En tal caso, el número de búsquedas a realizar no sería función lineal del tamaño de la tabla, ya que:

- El número máximo de entradas a rastrear coincide con el número de componentes léxicos reconocidos en el análisis, no con el número de caracteres almacenados en el texto como en el caso de asociación indirecta por posición.
- Las claves de acceso a la TTLT son un subconjunto de los números naturales comenzando en 1 y terminando en el número correspondiente al del mayor componente léxico reconocido. La mayor parte de los elementos de ese conjunto se corresponden con entradas válidas.
- Los TTR están ordenados lexicográficamente en función del campo o campos que almacenan las coordenadas del carácter inicial. Esto significa que el TTR para la posición i de la TTLT almacenaría una posición que sería lexicográficamente inferior que la almacenada por el TTR correspondiente con la entrada $i + 1$. El orden lexicográfico se establece dando mayor peso a las líneas que a las columnas, de tal modo que un TTR con línea y será menor que cualquier TTR con línea y' tal que $y' > y$. En el caso de que $y = y'$, el TTR menor será aquel que posea una columna menor. No se puede dar el caso de que dos TTR's tuviesen la misma posición ya que un carácter sólo puede pertenecer a un componente léxico. Por tanto la relación de orden sería total.

Entonces es posible realizar una búsqueda dicotómica o binaria sobre la TTLT utilizando como elemento de búsqueda la posición señalada por el usuario y finalizando cuando se encuentre un TTR tal que sus caracteres asociados incluyen al localizado en dicha posición. Por tanto, el tiempo medio y máximo de búsqueda es $log_2(n)$, donde n

[13] Nótese que en la asociación indirecta por posición, al modificar un componente léxico era preciso buscar las entradas de la TTLT para todos los *caracteres* incluidos en el ámbito que se viera afectado por la modificación, que podía ser el resto de la línea o el resto del texto dependiento de la existencia o no de modificación en el número de líneas del texto del componente léxico.

representa el número del mayor componente léxico reconocido. El problema planteado por la posibilidad de que una entrada de la tabla en el rango de 1 a n esté vacía se puede solucionar fácilmente accediendo al siguiente (o al anterior) elemento de la tabla con entrada no vacía.

Sin embargo, no es deseable almacenar las coordenadas de inicio de cada componente léxico puesto que ello supondría la reaparición del problema de los puntos de sincronización para mantener consistente la información de los TTR con el texto, con lo que volverían a surgir los problemas estudiados en la asociación indirecta por posición, sólo que esta vez, en lugar de mover los TTR de una entradas de la TTLT a otras, se actualizarían campos dentro de los propios TTR's. En resumen, añadiendo las coordenadas de inicio del texto de cada componente léxico a los TTR, lo que se consigue es tener todos los defectos de la asociación por posición aumentados con una mayor complejidad de acceso.

2.5.2 La estructura de representación de los componentes léxicos

Para representar los TTR's se pueden utilizar vectores que almacenen al menos la siguiente información:

- El desplazamiento del inicio del texto del componente léxico con respecto al inicio del texto del componente léxico precedente.

- La longitud del texto asociado al componente léxico.

Adicionalmente, se podría almacenar en cada TTR información de análisis relacionada con el componente léxico.

Cuando el usuario pulse con el ratón sobre un carácter del texto y esté activo una de las operaciones de edición de componentes léxicos, se debe llamar a la función `search-ttr`, la cual se encargará de determinar el TTR al que pertenece dicho carácter utilizando para ello un algoritmo de búsqueda lineal, como se describió anteriormente. Para cada TTR se debe calcular la coordenada final de su texto para saber si el carácter señalado por el usuario se encuentra dentro del ámbito de ese TTR. Debido al carácter general que debe poseer el editor de componentes léxicos, hay que considerar el caso de que un componente léxico tenga asociado un texto de varias líneas, lo cual influirá en el modo en que se calculen las coordenadas finales de cada componente léxico, puesto que la simple suma de la longitud a la posición inicial no dará un resultado válido en todos los casos.

El mantenimiento de la consistencia entre la información almacenada en la TTLT y el texto contenido en el editor es mucho más fácil que en el caso de la asociación indirecta por posición, puesto que basta con hacer que la función invocada cuando se produce un evento de teclado determine si la acción realizada consiste en borrar o en insertar un carácter y en consecuencia disminuirá o aumentará en uno la longitud del texto en el TTR. Por consiguiente se puede utilizar una única función para realizar las tareas que en la estrategia anterior realizaban `insert-character` y `delete-character`. En el caso de la inserción de un carácter, dicha función se encargará además de que su representación en pantalla se corresponda con la fuente y los colores correspondientes a la operación de edición de componentes léxicos que se esté llevando a cabo.

2.6 Conclusiones sobre la asociación indirecta

La asociación indirecta por desplazamiento presenta como principal ventaja respecto a la asociación indirecta por posición la existencia de una mayor independencia entre las operaciones propiamente de edición del texto y las tareas relacionadas con el análisis.

Su principal inconveniente radica en la necesidad de realizar una búsqueda lineal para determinar el componente léxico sobre el cual se va a relizar la operación de edición solicitada por el usuario. Recordemos que en la asociación por posición dicho acceso se realiza directamente.

A la hora de balancear ventajas e inconvenientes para establecer la estrategia a utilizar, se deben tomar en cuenta el siguiente hecho, que se va a dar como norma general en la mayoría de los casos: los caracteres de salto de línea raramente se van a considerar como caracteres válidos dentro de un componente léxico.

Normalmente los caracteres de cambio de línea van a actuar de separadores. Por tanto, no se puede dar el caso de necesitar una reorganización de la tabla de enlace entre los componente léxico y el texto para todas las entradas que se refieran a texto situado a continuación del correspondiente a un componente léxico que ha variado su número de líneas. Como consecuencia, desaparece la causa del elevado consumo de recursos computacionales atribuido a la asociación por posición. En esta situación, dado un número t de componentes léxicos, con una logitud media de c caracteres por componente léxico en un texto de l líneas, se tendría que como media:

- El algoritmo de asociación por desplazamiento precisaría de $t/2$ operaciones de acceso a la tabla hash para determinar el componente léxico sobre el que operar.
- El algoritmo de asociación por posición precisaría de:
  - Un número de accesos a tabla hash igual a $c/2$ para encontrar el componente léxico sobre el que operar.
  - Un número $(t \times (c+1))/(l \times 2)$ de operaciones de acceso a la tabla hash para actualizar los TTR de la línea afectada por las modificaciones en el componente léxico editado[14].

Por tanto, la asociación por posición requerirá un menor número de accesos a la TTLT cuando se cumpla que:

$$\frac{t \times (c+1)}{l \times 2} + \frac{c}{2} < \frac{t}{2}$$

Consideremos como ejemplo un texto 100 líneas, en el que se han reconocido 800 componentes léxicos y la longitud media del texto de cada componente léxico es de 10 caracteres. Tenemos entonces que $t = 800, c = 10$ y $l = 100$. Por consiguiente el número de accesos será:

- $\frac{t}{2} = \frac{800}{2} = 400$ accesos en la asociación por desplazamiento.
- $\frac{t \times (c+1)}{l \times 2} + \frac{c}{2} = \frac{800 \times (10+1)}{100 \times 2} + \frac{10}{2} = 44 + 4 = 48$ accesos en la asociación por posición.

[14] $(c+1)$ representa la longitud media del texto de un componente léxico más el carácter de separación con el siguiemte. $t \times (c+1)$ representa el número total de caracteres en el texto. Si lo dividimos por l, la longitud media de cada línea, obtenemos el número medio de caracteres por línea. Por tanto, el número medio de entradas a reorganizar en la TTLT será de $(t \times c)/(l \times 2)$

Por tanto se observa claramente que la estrategia de asociación indirecta por posición es mejor en los casos en los que no se permiten caracteres de ruptura de líneas en el texto de los componentes léxicos.

2.7 Asociación multinivel

En esta sección se va a realizar el análisis de una nueva estrategia que desarrolla los conceptos de la asociación indirecta extendiéndolos más allá de la indirección simple para construir estructuras complejas de asociación con varios niveles de profundidad.

Mientras que en las estrategias de *asociación indirecta simple*[15] se calcula en un único paso la clave de acceso a la estructura de representación del componente léxico mediante la realización de una serie de transformaciones sobre la posición absoluta o relativa del texto correspondiente, en la presente estrategia el paso del texto al componente léxico se realizará en varias etapas, cada una de ellas conducente a una estructura que representará la asociación de un conjunto de texto[16] con un conjunto de componentes léxicos.

Conceptualmente esto equivale a construir una estructura arborescente para enlazar el texto con los componentes léxicos, de modo que la realización de cada una de las etapas de indirección equivale a avanzar un nivel de profundidad en el árbol. El proceso de indirección puede verse como una búsqueda en profundidad en dicho árbol tal que la información disponible nos permite determinar en cada nodo el camino correcto en que debe avanzar la búsqueda.

2.7.1 La estructura de enlace

La parte central de esta estrategia la constituye la estructura de datos, con forma de árbol, que se utilizará para establecer los enlaces entre el texto y los componentes léxicos. Desde el punto de vista lógico, dicha estructura se caracteriza por:

- Cada nodo del árbol representa a la vez:
  - un conjunto de texto lógicamente relacionado[17]
  - el conjuto de componentes léxicos resultado del análisis de dicho texto.
- La raíz del árbol representa todo el texto contenido en el editor de componentes léxicos y por tanto también todos los componentes léxicos resultado de su análisis.
- Las hojas del árbol representan la asociación de un único componente léxico con su texto correspondiente.

Pasando ya a aspectos más prácticos, desde el punto de vista de la eficiencia computacional, debe poseer la suficiente flexibilidad como para permitir la realización de operaciones de inserción, modificación y borrado de nuevos componentes léxicos sin que ello suponga la realización de una costosa reorganización sobre una parte significativa del árbol.

[15] Con este término nos referimos tanto la asociación indirecta por posición como la asociación indirecta por deplazamiento.
[16] El texto correspondiente a 0, 1 o varios componentes léxicos
[17] En la práctica dicho texto será contiguo, aunque se podría tratar con partes discontinuas de texto.

Utilización de B árboles

Una estrcutura de datos que se adapta muy bien a estas características es la correspondiente a un B^+ árbol o a un B árbol. Se puede encontrar una descripción de las características de ambos tipos de árboles en [Korth y Silberschatz 87].

Las ventajas que presenta la utilización de un B^+ árbol son las siguientes:

- Todos los nodos del árbol tienen la misma estructura, lo cual facilita la escritura de la función de búsqueda.
- Todas las ramas tiene la misma longitud, por lo que el tiempo de acceso a las hojas es igual para todas ellas.

Los B árboles, por su parte, presentan como ventajas más significativas:

- Eliminan el almacenamiento redundante de claves de búsqueda.
- No todas las ramas son de la misma longitud, sino que, en función de la distribución de los valores de las claves de búsqueda, ciertos elementos pueden ser accedidos directamente desde niveles superiores del árbol.

Genéricamente, estas ventajas de los B árboles se ven ensombrecidas por los siguientes inconvenientes:

- La gestión y el mantenimiento del índice se complica, pues es necesario manejar dos tipos de nodos distintos[18].
- Las operaciones de eliminación en un B árbol son más complejas, puesto que los valores eliminados no residen necesariamente en las hojas.

Se podría asumir la mayor complejidad asociada a la gestión de B árboles con el fin de diseñar una estructura de almacenamiento más compacta, aunque la utilización dual de los nodos intermedios[19] suponga una disminición de la simplicidad y claridad que proporciona la estructura B^+ árbol

Utilización de claves segmentadas

La utilización de B o B^+ árboles impone la necesidad de utilizar la clave completa para etiquetar los nodos, de modo que la función de búsqueda siempre utiliza la totalidad de la clave en todos los niveles del árbol.

En este trabajo se propone la utilización de claves segmentadas para optimizar el proceso de búsqueda y la organización de la estructura arborescente. En cada nodo se utilizará el siguiente segmento de la clave como criterio de decisión para determinar el camino a seguir.

Como ya se expuso en la discusión de las otras estrategias, la interación del usuario con el editor de componentes léxicos se realiza directamente mediante pulsaciones sobre el texto, puesto que éste es el método más intuitivo y práctico para el usuario. Esto

[18] En un B árbol la estrucutura de almacenamiento utilizada por los nodos internos es diferente a la utilizada por los nodos hoja.

[19] Los nodos intermedios se utilizan simultáneamente como enlaces con los nodos de nivel inferior y como enlaces a los datos finales, en este caso los componentes léxicos.

conlleva la necesidad de utilizar la posición de un carácter significativo del texto de cada componente léxico, generalmente el primero, como clave de acceso a la estructura de asociación con los componentes léxicos. Por consiguiente, el único método natural de segmentar la clave consiste en descomponerla en los valores de las coordenadas componentes, esto es, considerar cada clave como la composición del valor de la coordenada horizontal con el valor de la coordenada vertical.

El proceso de búsqueda utilizará la coordenada horizontal (el número de línea) para indexar el nodo raíz y obtener el nodo que representa los componentes léxicos de dicha línea, para a su vez indexar dicho nodo con el valor de coordenada vertical y obtener así el componente léxico cuyo texto comienza en esa posición.

Por consiguiente se utilizará una estructura de dos niveles de indexación, en la que el primer nivel permite discrimar el conjunto relevante de entradas del segundo nivel, las cuales apuntan directamente a los datos.

2.7.2 El árbol de enlace componente léxico-texto

La estructura utilizada para representar la asociación entre el texto y los componentes léxicos, que seguiremos denominando TTLT como en el caso de la asociación indirecta[20], va a tener la forma de un árbol de tres niveles:

- El primer nivel del árbol estará constituido por un solo nodo que almacenará una estructura de datos capaz de dirigir la búsqueda hacia el nodo adecuado del segundo nivel, utilizando como elemento de decisión el número de línea del texto cuyo componente léxico se trata de encontrar.

- El segundo nivel del árbol tendrá asociado un nodo por cada una de las líneas presentes en el texto. Una vez que el proceso de búsqueda ha llegado a uno de los nodos del segundo nivel, debe ser capaz de determinar, a partir únicamente de la columna en la cual comienza el texto, cuál es la estructura de representación del componente léxico asociada al texto.

- El tercer nivel del árbol estará constituido por nodos hoja. Existirá uno de estos nodos por cada componente léxico reconocido en el proceso de análisis del texto. Cada nodo almacenará el TTR correspondiente a un componente léxico.

Una vez definida la estructura general del árbol, queda por analizar la constucción de cada uno de los diferentes tipos de nodos.

El nodo raíz

Conseguir una implementación eficiente del nodo raíz es fundamentel si queremos conseguir que el editor de componentes léxicos sea capaz de realizar las funciones de búsqueda de componentes léxicos de manera eficiente y a la vez con la mayor economía posible de recursos computacionales.

El nodo raíz es importante por las siguientes causas:

[20] Realmente el significado de TTLT en el caso de la asociación multinivel debería ser el de **T**oken **T**ext **L**ink **T**ree (árbol de enlace componente léxico-texto). Sin embargo, para evitar confusiones, se ha decidido mantener el signicado de las siglas, puesto que sigue teniendo validez

- Puesto que su misión es la de actuar de enlace entre el primer y el segundo nivel del árbol, debe poseer estructuras de asociación que permitan ligar cada número de línea (el segmento de clave utilizado en el primer nivel) con el nodo de segundo nivel correspondiente. Para textos grandes, el tamaño del nodo raíz puede ser muy grande. Contruir el nodo raíz de forma que utilice eficientemente la memoria es un factor importante de diseño.

- Como consecuencia de un tamaño potencialmente elevado, es importante que la estructura de datos mediante la cual se implemente este nodo permita un acceso efectivo igual de rápido para todos los elementos. No se pueden consentir retardos elevados en el acceso a ninguno de los niveles del árbol puesto que ello implicaría una disminución del rendimiento global de la función de búsqueda.

- El problema de rendimiento que surgía en las diferentes estrategias de asociación indirecta con aquellas modificaciones de los componentes léxicos que suponían la eliminación o incorporación de caracteres de ruptura de líneas, puede ser evitado en gran parte en la estrategia multinivel mediante la implementación del nodo raíz utilizando una estructura que permita realizar facilmente actualizaciones sin necesidad de copiar realmente todo el contenido de los subárboles afectados.

Como siempre, a la hora de elegir la estructra de datos mediante la cual se implementa el nodo raíz, podemos optar entre un cierto número de alternativas. Entre ellas tenemos:

- Listas, que no son adecuadas debido a que el tiempo de acceso a los elementos no es constante, si no que varía en función de su posición en la lista.

- Conjuntos, en principio más adecuados debido a la unicidad de las claves utilizadas, presentan los mismos inconvenientes que las listas.

- Vectores, inadecuados debido al carácter estático de su tamaño.

- Tablas hash, ya utilizadas anteriormente en las estrategias de asociación indirecta, se vuelven a presentar como la opción preferida ya que presentan las siguientes ventajas:

  - Proporcionan un tiempo de acceso constante para todos los elementos, independientemente del valor de la clave.
  - Permiten realizar fácilmente inserciones y borrados.
  - Permiten una rápida reorganización en caso de que sea necesario desplazar las entradas de la clave como consecuencia de la modificación de los caracteres de rupturas de líneas en el texto de alguno de los componentes léxicos[21].
  - Independizan el nodo raíz de los nodos intermedios. Esto se debe a que las tablas hash almacenan punteros, no valores en sí, por lo que puede modificarse la estructura apuntada con total libertad.

Por tanto, se utilizarán tablas hash para construir el nodo raíz del árbol de enlace.

[21] En la estrategia de asociación indirecta por posición, el desplazamiento de las entradas en la tabla suponía un gran consumo de recursos debido a que no se conocían a priori las claves de acceso a los componentes léxicos. En el caso presente, sí se conocen los números de líneas que han de ser modificados, por lo que el acceso a las entradas correspondientes es directo. Por consiguiente no es necesario rastrear todo el texto.

Los nodos internos

Una vez que se ha diseñado la estructura del nodo raíz se debe pasar a tratar los aspectos relacionados con los nodos de la siguiente capa del árbol, los nodos internos.

Estos nodos tienen como misión proporcionar un enlace eficiente entre el segundo y el tercer nivel del árbol mediante la asociación del siguiente segmento de la clave (el número de columna) al elemento de representación del componente léxico correspondiente al texto cuyo primer carácter tiene por coordenada vertical el primer segmento de la clave y por coordenada horizontal el segundo segmento.

Las consideraciones que se barajan en el caso de estos nodo son diferentes a las que se utilizaban en el caso del nodo raíz, puesto que ahora el tamaño no es una característica dominante[22]. Además, ahora no se trata de establecer la mejor estructura para un único nodo, como en el caso del raíz, sino que se debe buscar una estructura eficiente de la que existirán múltiples instancias, una por cada nodo intermedio, esto es, una por cada línea presente en el texto, teniendo en mente que su número puede puede ser potencialmente elevado.

Una vez más examinamos las diferentes alternativas:

- La utilización de tablas hash no parece ser en este caso la opción más favorable, por dos razones principalmente:
  - El pequeño tamaño de cada una de las líneas no hace necesario la utilización de una estructura de almecenamiento compleja. El reducido número de componentes léxicos por línea puede hacer que la el tiempo medio de búsqueda por clave sea más lento que una simple exploración secuencial.
  - Al existir una tabla por cada línea sería necesario reservar almacenamiento para cada una de ellas. La gestión de este almacenamiento por parte del garbage collector encarece el uso de este tipo de estructuras.
- Las listas se presentan como alternativa más viable, una vez descartados los vectores por su conocida limitación del tamaño fijo. Las listas de pequeño tamaño se muestran realmente como estructuras muy eficientes, sobre todo si se utilizan las funciones que modifican físicamente los elementos de las listas[23]. Puesto que los elementos de la lista deberán ser pares (posición,componente léxico), parece adecuado utilizar una lista de asociación para cada nodo intermedio, de modo que dicha lista contenga las asociaciones entre el segundo segmento de la clave de un componente léxico y la estructura de representación del componente léxico propiamente dicho.

Por tanto, se utilizarán listas de asociación para construir los nodos intermedios del árbol de enlace.

Los nodos hoja

Los nodos hoja son aquellos nodos del árbol de enlace que en se encuentran al final de los caminos de búsqueda, o lo que es lo mismo, al final de las distintas ramas del árbol.

[22] Mientras que un texto puede tener miles de líneas, cada línea contendrá un número reducido de componentes léxicos, típicamente menos de una decena.

[23] Esta característica se presenta muy útil en nuestro caso, puesto que la operación más común, aparte del acceso a los elementos, será la reasignación de posiciones en la línea por el desplazamiento provocado por la modificación del texto de un componente léxico.

La misión de un nodo hoja es almacenar la información de análisis relativa a un componente léxico que pueda resultar de interés para la interacción con el usuario o que pueda resultar necesaria para realizar las operaciones de edición del texto de los componentes léxicos. Es fácil observar la similitud entre las funciones realizadas por los nodos hoja y las correspondientes a los TTR de la estragia indirecta. Ciertamente tal similitud no es casual, pues las diferentes variantes de la asociación indirecta pueden verse como casos simplificados de la asociación multinivel en las que tan sólo existe un nivel de indirección. Según esto, las distintas variantes se centrarían en encontrar una forma adecuada de representar su pequeño árbol de anlace.

En lo que difiere la asociación multinivel es en la introducción de múltiples niveles de indirección, con lo que se aumenta la distancia entre la raíz del árbol[24] y las hojas. Por eso es lógico que las TTR no presenten más que pequeñas variaciones[25]. Puede ser incluso significativo comprobar como en el nodo raíz se sigue manteniendo una tabla hash como estructura de representación.

2.7.3 La estructura `ICEeditor`

La estructura `ICEeditor` es la estructura de datos central de ICEeditor, en torno a la cual se organiza el programa. Se utiliza para establecer una relación estructurada entre los siguientes elementos:

- La imagen de la interfaz gráfica de usuario, en la que se incluye el texto del editor de componentes léxicos así como los elementos auxiliares (botones, menús, editores de líneas, etc) necesarios para establecer una adecuada interacción con el usuario.
- El árbol de enlace componente léxico-texto, que como se ha visto contiene la información necesaria para establecer un enlace eficiente entre la representación textual de cada componente léxico y la información provimiente de su análisis léxico-sintáctico.
- Información de las operaciones realizadas en el editor de componentes léxicos. En este apartado se incluye tanto la información sobre la operación actual que se está llevando a cabo como información global relativa a las modificaciones realizadas sobre los componentes léxicos desde el último análisis realizado sobre el texto.

En este capítulo no se trata la información concerniente a la imagen de ICEeditor, por lo que la discusión se centrará en aquellos campos relacionados directamente con el comportamiento interno del editor de componentes léxicos.

El campo `TTLT`

Este campo almacena el árbol de enlace. Realmente es un puntero de entrada al nodo raíz, a partir del cual podremos acceder a cualquiera de los nodos hoja, pasando por el nodo intermedio correspondiente.

Se ha utilizado una tabla hash que utiliza `eq` como predicado de comparación para el acceso a la clave. La elección de este predicado en lugar de `equal` se debe a la utilización

[24] En la asociación indirecta la raíz del árbol la constituiría la propia TTLT.

[25] Relativas a los datos dependientes del texto que deben almacenar (posición, longitud, ...), no a los datos del análisis léxico-sintáctico en ellos almacenados.

del número de línea, que es un entero, como clave. El predicado `eq` se muestra como el más eficiente predicado de comparación entre enteros. Recordemos que la función de la tabla hash incorporada en el nodo raíz es la de actuar de puente hacia los nodos intermedios, como un índice de primer nivel construido a partir de los números de línea.

El campo `operations`

La interacción del usuario con los componentes léxicos es un proceso dinámico, en el que sucesivamente se van realizando operaciones de edición sobre los componentes léxicos, incluyendo la posibilidad de editar varias veces el texto correspondiente a un mismo componente léxico antes de proceder a realizar un nuevo análisis del texto.

Cuando se realice un nuevo análisis será preciso pasarle a ICE información sobre los componentes léxicos que han sido editados. Es por ello que se necesita algún lugar en el que ir almacenando datos acerca de las modificaciones realizadas entre dos análisis consecutivos.

Para almacenar esta información se utiliza el campo `operations`. Cada vez que se termine una operación de edición sobre un componente léxico, se almacenará en este campo, en la entrada correspondiente a ese componente léxico, el tipo de operación realizado, el texto antiguo y el nuevo texto editado. Puesto que toda operación de edición puede verse como una operación de modificación del componente léxico[26], es conveniente transformar cada operación realizada en su modificación equivalente antes de almacenarla en el campo `operations`.

La utilización de este campo permite al implemetador optar entre las siguientes alternativas a la hora de tratar con la edición de componentes léxicos borrados[27]:

- Dar un mensaje de error cuando el usuario intente editar un componente léxico borrado. Ello obliga a acceder a este campo siempre que se desea editar un componente léxico para comprobar que no ha sido eliminado por una operación de borrado. Si además se desea distinguir entre un borrado *explícito*, realizado utilizando la operación `delete` de un borrado *implícito*, realizado mediante el borrado de todos los caracteres del componente léxico durante una operación `modify`, se deberá almacenar en cada entrada de esta tabla un elemento que indique si ha sido borrado o no.

- Permitir la operación de edición partiendo del hecho de que el texto del componente léxico se corresponde con la cadena vacía.

Se ha optado por utilizar una tabla hash porque proporciona un método de acceso rápido, con tiempo constante y evita el tener que realizar las ordenaciones a las que nos veríamos obligados en el caso de haber optado por otras estructuras, ya que el usuario posee libertad total en la elección del orden en que modificará los componentes léxicos. Puesto que los elementos de esta tabla tendrán un número fijo de componentes, lo más adecuado es utilizar vectores para su representación, con lo que evitamos el elevado coste que supone la utilización de elementos de tipo estructura.

[26] Una operación de inserción puede verse como añadir el nuevo texto al principio del siguiente y una operación de borrado como una modificación en la que el nuevo texto es la cadena vacía.

[27] Esto es, qué hacer cuando el usuario pretenda realizar una operación de edición que no sea `info` sobre un componente léxico que había sido borrado previamente por el usuario.

El campo `parsered-p`

Este campo tiene caracter lógico, es decir, sus posibles valores son `()` (falso) y `t` (verdadero). Se le ha añadido el sufijo `-p` porque el método utilizado para su recuperación se comporta como un predicado lógico. El significado de este campo es el siguiente:

- Si no se ha realizado ningún análisis sobre el texto, entonces contendrá el valor `()`. Esto significa que el usuario podrá actuar libremente sobre el texto, puesto que aún no se ha establecido sobre él la estructura de componentes léxicos. No se realiza por tanto ningún acceso a la TTLT ni se almacena información de operación.
- Si el texto ya ha sido analizado alguna vez, entonces contendrá el valor `t`. Con ello se indica que ya ha sido establecida una estructura de componentes léxicos que regirá en lo sucesivo sobre cualquier modificación que se pretenda realizar en el texto. A partir de este momento se realizarán accesos a la TTLT y se almacenará la información de operación.

El campo `edit-state`

Este campo indica la operación actual que se está realizando o se va a realizar sobre alguno de los componentes léxicos. El valor de este campo podrá ser:

- `'insert` si la operación que se va a llevar a cabo es la inserción de un componente léxico.
- `'delete` si se va a eliminar un componente léxico.
- `'modify` si se va a modificar el texto de un componente léxico ya existente.
- `'info` si el usuario tan sólo desea acceder a la información almacenada en el TTR correspondiente a un componente léxico.
- `()` si el texto aún no ha sido analizado.

Según el valor almacenado en este campo, la información de edición será actualizada de forma distinta, y se utilizarán los colores y fuentes apropiados a la operación correspondiente.

El campo `current-op`

Este campo es muy útil en las tareas de sincronización del texto almacenado en el editor con la información presente en el árbol de enlace. Al igual que en la estrategia indirecta, en la estrategia multinivel es adecuado mantener puntos de sincronización en los cuales se garantiza la perfecta coherencia entre la información de la TTLT y el texto, mientras que entre dos puntos de sincronismo consecutivos se permiten modificaciones controladas sobre el texto. Con ello se evita la sobrecarga computacional que implicaría el tener que ir actualizando la TTLT cada vez que se inserta o borra un carácter.

Uno de los acontecimientos que dispara la activación de un punto de sincronización es la elección de una nueva operación de edición. En ese momento el campo `edit-etate` cambiará de valor para reflejar la nueva operación. Sin embargo, los cálculos de la

sincronización deberán realizarse teniendo en cuenta la operación realizada sobre el último componente léxico, es decir, la que se halla precisamente almacenada en este campo. Una vez realizada la sincronización, este campo pasará a tener el mismo valor que `edit-state`.

2.7.4 Acceso al los TTR

El acceso a los componentes léxicos es una acción dirigida por el usuario, ya que cuando éste pulsa con el ratón sobre el texto y el editor de componentes léxicos se encuentra en uno de los estados de edición (campo `edit-state` con valor distinto de `()`), es necesario acceder al TTR del componente léxico a cuyo texto pertenece el carácter señalado, puesto que se precisa de la información en él almacenado para guiar la correcta realización de la operación de edición del componente léxico.

Para que sea posible lograr este tipo de acceso hay que determinar la posición (fila y columna) del carácter señalado con el ratón. Para ello puede ser necesario capturar el evento asociado a la pulsación de uno de los botones del ratón.

Sin embargo, la posición del carácter en el cual se pulsó el ratón puede que no constituya una clave válida de acceso a un TTR. Esto se debe a que cada TTR se indexa en el árbol de enlace tomando como referencia las coordenadas del primer carácter de su cadena de texto asociada. Por consiguiente, hay que utilizar un enfoque de búsqueda tentativa a partir de la posición de un carácter. Para ello se puede definir un método asociado a `ICEeditor` llamado `search-ttr-backward`. Este método deberá acceder al nodo intermedio mediante el número de línea y tratar de buscar un carácter que sea el inicial de un componente léxico y cuya columna sea menor o igual que la pasada como argumento. Para ello bastará con recuperar los pares de asociación de la lista e ir comparándolos con la clave. Si la búsqueda no tiene éxito en esa línea, deberá repetir el proceso para la línea anterior.

Si el inicio del componente léxico se encuentra en la misma línea (que es el caso normal) el proceso de búsqueda resulta computacionalmente barato puesto que se trata de realizar accesos a una lista, estructuras bien manejadas por LISP.

2.7.5 Sincronización del árbol de enlace

Debido al coste que supondría actualizar constantemente el árbol de enlace cada vez que se inserta o se borra un carácter, es interesante mantener el concepto de puntos de sincronización tal como se estableció en la estrategia de asociación indirecta. La variación que presenta la asociación multinivel a este respecto es una mayor eficiencia en la operación de sincronización. Las razones de este mejor comportamiento se manifiestan principalmente en el caso de que se inserten o se borren en el texto caracteres de salto de línea, puesto que ya no es necesario buscar en la TTLT todos los caracteres del texto para reasignar las claves.

Al utilizar claves particionadas, conseguimos independizar la posición horizontal de un componente léxico de la línea en la cual está. De este modo, para mover las líneas tan sólo es necesario modificar las entradas en el nodo raíz, manteniendo invariables los nodos intermedios, excepto claro está, la última línea del componente léxico del texto del componente léxico editado, en la que sí se deberán realizar ajustes.

Resuminedo, tenemos que la realización del proceso de sincronización conlleva:

- Reorganizar una sóla línea (un nodo intermedio) si no se han introducido o eliminado caracteres de salto de línea.

- Reasignar los punteros de la tabla hash del nodo raíz y reorganizar la lista de un nodo intermedio en el caso de que se produzcan modificaciones en los caracteres de ruptura de líneas. Como cada línea deberá tener una entrada en el nodo raíz, no se realiza ningún acceso innecesario a dicha tabla hash y puesto que se pueden recuperar todos los pares de asociación de la lista intermedia, tampoco se realizarán accesos innecesarios en los nodos intermedios.

2.8 Conclusiones

Podemos realizar un pequeño análisis cuantitativo sobre el rendimiento de cada estrategia. Consideremos un texto de t componentes léxicos, con una longitud media de c caracteres por componente léxico en un texto de l líneas. En el caso de la modificación de caracteres de salto de líneas:

- En la estrategia de asociación indirecta por posición se precisarían $\frac{t}{l} \times \frac{l}{2} \times c$ accesos a la TTLT para realizar la sincronización, de los cuales sólo $\frac{100}{c}$ serán acceso válidos a TTR[28].

- En la estrategia de asociación multinivel se precisan $\frac{t}{2}$ accesos al nodo raíz y $\frac{t}{2 \times l}$ para reorganizar un único nodo intermedio.

Tomemos como ejemplo un texto de 100 líneas en el que se han reconocido 800 componentes léxicos con una longitud media de texto de cada componente léxico igual a 10 caracteres. Tenemos entonces que $t = 800$, $c = 10$ y $l = 100$. Por consiguiente el número de accesos será:

- $\frac{t}{l} \times \frac{l}{2} \times c = \frac{800}{100} \times \frac{100}{2} \times 10 = 4000$ en el caso de la asociación indirecta por posición.

- $\frac{t}{2} + \frac{t}{2 \times l} = \frac{800}{2} + \frac{800}{2 \times 100} = 400 + 4 = 404$ accesos en el caso de la asociación multinivel.

Se puede observar que la diferencia viene marcada por el número de accesos innecesarios que se realizan en la asociación indirecta por posición, ya que por cada componente léxico se accede a c elementos de la tabla hash. Por contra, en la asociación multinivel no se realizan accesos innecesarios, por lo que no es de extrañar que divida por c la cantidad de accesos a realizar.

Con respecto a la asociación indirecta por desplazamiento, que no precisaba reorganizar la TTLT después de la edición de un componente léxico, debemos considerar la velocidad de acceso:

- En la asociación por desplazamiento se precisan $t/2$ acceso a la tabla hash para determinar el componente léxico sobre el cual se va a actuar.

- En la asociación multinivel tan sólo se precisa un acceso a la tabla hash y $\frac{t}{l \times 2}$ accesos a la tabla de asociación.

[28] Los correspondientes a las posiciones de comienzo del texto de los componentes léxicos.

El tiempo de acceso a cada TTR en la asociación multinivel es constante e inferior al de la asociación por posición cuando hay más de una línea.

La asociación multinivel es superior a la asociación por posición en cualquier caso. Si no se producen manipulaciones en los caracteres de ruptura de líneas, la asociación multinivel es netamente superior a las asociación por desplazamiento. Con la modificación de las líneas, la asocición multinivel minimiza el número de accesos necesarios para reorganizar el árbol de enlace.

Un factor muy importante es que la asociación multinivel mantiene constante el tiempo de acceso a un TTR, mientras que en la asociación indirecta por desplazamiento el rendimiento se va degradando conforme aumenta el número de componentes léxicos.

2.9 Direcciones futuras

Si desde un punto de vista cuantitativo la asociación multinivel muestra unas muy buenas cualidades, quizá sea más importante destacar los aspectos cualitativos relacionados con su capacidad de ampliación futura.

Una dirección interesante en la que quizá debería evolucionar el árbol de enlace es hacia una mayor integración con el analizador sintáctico para que su estructura sea capaz de representar la asociación, no sólo de los componentes léxicos con el texto, sino también de estructuras sintácticas de mayor nivel con el texto subyacente.

Desde este punto de vista se puede decir que el árbol de enlace representa actualmente una estructura sintáctica artificialmente impuesta en la que se organizan los componentes léxicos en líneas. La gramática podría ser la siguiente:

```
text ::= <end-of-text>
      |  line text

line ::= <end-of-line>
      |  token line
```

Avanzando un paso más en la integración del editor de componentes léxicos con el proceso de análisis, se podría sustituir esta gramática artificial por la gramática propia del texto analizado. Ello implicaría también ampliar el conjunto de operadores a disposición del usuario, ya que precisaría al menos de una operación *focus* y de una *unfocus* para poder moverse ascendente y descendentemente por la jerarquía del árbol de enlace impuesta por la gramática. Tentativas en esta dirección han sido ensayadas en el sistema de *pretty-printers* presente en CENTAUR [Centaur 92a, Centaur 92b, Centaur 92c].

Capítulo 3

ICEeditor según AÏDA

En este capítulo se pretende realizar una descripción de ICEeditor. El propósito de esta herramienta es salvar el espacio existente entre el analizador generado por ICE y el usuario, de modo que se produzca un acercamiento lo más intuitivo posible por parte de este último a las características de ICE, permitiendo sacar el mayor provecho posible de su carácter incremental.

Para conseguir tales objetivos se ha dotado a ICEeditor de las siguientes capacidades:

- Una interfaz de usuario basada en representaciones gráficas estándar que facilitan la interacción con el usuario y aumentan el confort de éste al encontrase con elementos que ya le son familiares por su amplia utilización en los entornos de ventanas comercialmente disponibles.
- Elevadas prestaciones en la edición de componentes léxicos, basadas en la consecución de una eficiente asociación entre los componentes léxicos y el texto, mientras se consigue aislar al usuario de la complejidad subyacente.
- Interacción dinámica entre la interfaz gráfica, el analizador léxico y el analizador sintáctico, con lo cual es posible integrar de un modo altamente consistente los tres elementos al mismo tiempo que se dota al sistema de una elevada modularidad, puesto que que cada componente puede ser modificado independientemente de los demás.

En este capítulo se van a tratar principalmente aspectos relativos a la interfaz gráfica y a la interacción con el usuario. Los aspectos concernientes a la asociación entre los textos y su representación textual son tratados en el capítulo 2, mientras que la interacción con los analizadores léxico y sintáctico se trata en los capítulo 5 y 6, respectivamente.

3.1 Los componentes de ICEeditor

Desde el punto de vista de la interfaz gráfica, ICEeditor consta de los siguientes elementos, que forman la imagen asociada al objeto:

- Un *editor de componentes léxicos*, construido sobre la base de un editor de textos al que se le han añadido capacidades de edición de componentes léxicos mediante la incorporación de una estructura de datos compleja que permite manejar la asociación

entre un componente léxico y su texto correspondiente. Las operaciones permitidas en este editor se estructuran en función de los componentes léxicos. El editor de componentes léxicos es el elemento fundamental de ICEeditor, en torno al cual se articulan los demás elementos de la imagen.

- Un desplazador [1] constituido por barras de desplazamiento horizontales y verticales que permiten desplazar convenientemente el contenido del editor de componentes léxicos.

- Un *menu* mediante el cual el usuario puede seleccionar tanto las acciones a realizar sobre los componentes léxicos como aspectos relativos al comportamiento de ICEeditor, como puede ser el idioma utilizado, los colores mediante los cuales se destacan las diferentes operaciones, etc.

- Una *barra de botones* que permiten acceder rápida e intuitivamente a las opciones del menú más utilizadas.

Estos son los elementos básicos. Adicionalmente se puede disponer de otros elementos que faciliten la labor del usuario o mejoren su aceptación del sistema. Por ejemplo, se puede utilizar un pequeño editor de líneas siempre accesible mediante el cual establecer el nombre del fichero que se está editando. Sin embargo, es conveniente no recargar excesivamente el aspecto de cualquier aplicación puesto que ello distrae la atención del usuario, llegando incluso un exceso de estímulos a dificultar lo propia interacción usuario-aplicación.

3.2 El editor de componentes léxicos

Se va a tratar ahora sobre la construcción del principal componente de ICEeditor, el editor de componentes léxicos, mediante el cual se pretende conseguir una interfaz eficiente y consistente en entornos incrementales. Este editor será el encargado de realizar la mayor parte de las tareas involucradas en la interacción usuario-analizador y por tanto su correcto diseño es vital para conseguir una herramienta operativa.

3.2.1 Requerimientos

Un editor de componentes léxicos se puede ver como una versión transformada de un editor de textos habitual, de modo que permita la edición de un texto analizado de la forma más natural posible que sea compatible con el mantenimiento de la información precisada por el analizador para que éste sea capaz de realizar un post-procesamiento incremental eficiente de los cambios introducidos por el usuario. Para que esto sea posible, debe procederse a un cuidadoso análisis para determinar las funcionalidades que deberá proporcionar y las tareas que tendrá que realizar. Es muy importante tener en mente el impacto que tendrá en el resultado final cada uno de los requerimientos planteados, así como el esfuerzo computacional requerido por cada uno de ellos.

La lista de requerimientos para el editor de componentes léxicos es la siguiente:

[1] También conocido por su denominación en lengua inglesa, *scroller*.

- Al cargar un archivo de texto, mientras éste no sea analizado se deberá permitir editar su contenido como si se tratase de un editor normal. A tal efecto podrá:
  - Moverse libremente por todo el texto.
  - Insertar, borrar y modificar texo.
  - Realizar operaciones adicionales de edición (borrar una palabra, borral una línea, etc.)
  - Realizar búsquedas y sustituciones.
  - Cortar y pegar utilizando el ratón.

  Se dispondrá de combinaciones de teclas que permitan realizar rápidamente las operaciones descritas.

- Cuando el texto sea analizado, se estructurará mediante unidades elementales que se corresponderán con el texto asociado a cada uno de los componentes léxicos resultantes del análisis.

- En un texto analizado, ya no será posible realizar libremente operaciones normales de edición como las descritas en el primer punto. A partir del análisis, toda operación que suponga una modificación del texto deberá de ser realizada basándose en la nueva estructura de unidades textuales que se corresponden con los componentes léxicos. Las operaciones básicas permitidas serán:
  - Insertar texto entre dos componentes léxicos consecutivos.
  - Modificar el texto correspondiente a un componente léxico
  - Borrar el texto correspondiente a un componente léxico.
  - Obtener información de un componente léxico.

  La necesidad de este requerimiento estriba en que ICE debe conocer entre qué componentes léxicos se han realizado las modificaciones respecto al texto del análisis anterior para poder aplicar la incrementalidad al siguiente análisis del texto.

- El usuario podrá indicar mediante un simple click del ratón el punto en el que desea llevar a cabo alguna de las operaciones permitidas en un texto analizado. Se deberán utilizar recursos gráficos (fuentes, colores) que permitan distinguir fácilmente el alcance de la modificación[2].

Este conjunto de requerimientos expresa la funcionalidad básica que debe proporcionar el editor de componentes léxicos. Sin embargo, para conseguir un editor realmente útil es necesario formular un nuevo requisito concerniente a la eficiencia:

- El editor de componentes léxicos deberá realizar las operaciones requeridas en un tiempo lo suficientemente reducido como para no suponer una sobrecarga en el proceso de análisis incremental, entendiendo éste como la suma de los tiempos correspondientes tanto al análisis sintáctico propiamente dicho como los que se deben al procesamiento provocado por las operaciones de edición sobre los componentes

[2] El texto del componente léxico afectado en el caso de modificación o borrado o el nuevo texto añadido en el caso de una inserción.

léxicos. En este tiempo no se deben incluir los periodos asociados al mecanografiado ni a las operaciones relativas al texto que no involucren a las estructuras de representación del texto con respecto a los componentes léxicos.

Este requerimiento tiene una explicación evidente: de nada sirve tener un sistema de análisis incremental muy rápido, si al realizar una operación de edición, por ejemplo borrar un componente léxico, se provoca una carga de procesamiento tan elevada por la reorganización de las estructuras de datos que enlazan el texto con los componente léxico que llega a ser excesivamente significativa respecto al total e incluso perceptible por parte del el usuario: probablemente se emplearía menos tiempo realizando un reanálisis total mediante técnicas clásicas.

3.2.2 Opciones de implementación

Dentro de la librería de objetos que AÏDA proporciona, se encuentran varias clases diferentes de editores, que abarcan:

- Editores de líneas, objetos de tipo {`lineedit`}, que tan sólo operan sobre textos de una sóla línea. No trabajan bien con fuentes proporcionales.
- Editores de cadenas de caracteres, cuyo tipo es {`stringedit`}. Se caracterizan por ocupar poco espacio en memoria y por permitir el filtrado de caracteres. Son adecuados para introducir nombres de ficheros y tareas similares, contando con la ventaja sobre los editores de líneas de funcionar correctamente con fuentes proporcionales.
- Editores de texto con formato. El tipo de estos objetos es {`tokenedit`}, aunque la función de creación se denomina `formatedit`. Son un subtipo de {`stringedit`} que permite controlar que los caracteres introducidos cumplen ciertas características, como el que sean alfabéticos o numéricos, que el carácter que ocupa una determinada posición pertenece a un determinado conjunto de caracteres, etc.
- Editores de texto simple, pertenecientes a la clase {`medite`}. Mediante ellos se puede obtener un editor de texto básico con operaciones de edición al estilo del conocido editor Emacs. Como aclaración, señalar que la clase {`lineedit`} es una subclase de {`medite`}, por lo cual un editor de líneas tiene la misma potencia que un medite pero restringida a la edición de una única línea.
- Editores de texto mejorados, del tipo `textedit`. Este tipo de editores se incluyen en lo que se denomina *extensiones de AÏDA*. Los editores `textedit` permiten editar *texto adornado*, esto es, texto en diferentes colores y fuentes entre el cual se puede insertar cualquier tipo de aplicación AÏDA.

Todos estos editores se describen de forma detallada en [ILOG 92c], excepto los de tipo `textedit`, cuya descripción se encuentra en [ILOG 92a].

Para implementar el editor de componentes léxicos es preciso utilizar editores de tipo `textedit` ya que es el único tipo que permite la utilización de diferentes combinaciones de fuentes y colores dentro del mismo texto. Esta es una característica muy deseable en ICEeditor, puesto que permite al usuario identificar fácilmente los cambios introducidos

en los componentes léxicos desde el último análisis realizado. La limitación que, en este aspecto, presentan los editores `medite`, han forzado a que sean desechados como posibles candidatos aunque el resto de sus características se adaptasen bastante bien a los requisitos de ICEeditor.

3.3 La aplicación ICEeditor

ICEeditor va a constituir lo que en AÏDA se denomina una aplicación compleja, es decir, una aplicación construida a partir de elementos más simples, cada uno de los cuales es a su vez una aplicación AÏDA, y que se mantienen interrelacionados entre sí de tal modo que desde cualquiera de ellos es posible acceder a todos los demás.

En primer lugar se debe definir la estructura que almacenará los campos que contienen las propiedades utilizadas para gestinar la integración del texto con los componentes léxicos resultantes de las fases de análisis léxico y sintáctico. Para ello creamos un nuevo elemento en la jerarquía de objetos AÏDA. Puesto que ICEeditor va a ser una aplicación compleja que va a hacer uso de las aplicaciones AÏDA predefinidas pero que no puede englobarse dentro de ninguna de ellas, lo aconsejable es hacerlo depender directamente de la clase {`application`}.

El significado y la utilidad de los distintos campos de la estrucutura `ICEeditor` se explica detalladamente en la sección 2.7.3 de la página 27.

Para permitir la creación de aplicaciones de tipo `ICEeditor` se define también la función `create-ICEeditor`, la cual realiza una llamada al método `make` asociado a {`ICEeditor`} para posteriormente crear las subaplicaciones correspondientes y ligarlas entre ellas añadiendolas como componentes de `ICEeditor`.

3.4 Variables asociadas a ICEeditor

Para facilitar el trabajo con ICEeditor, se han definido una serie de variables de carácter global. Todas ellas han sido empaquetadas dentro de `#:ICEeditor`, con lo cual se eliminan las posibles interferencias que pudiesen surgir por la coincidencia de nombre con otras variables ya definidas. Estas varibles se refieren principalmente a:

- Definición de trayectorias.
- Definición de mensajes multi-idioma.
- Definición de cursores.
- Definición de fuentes.
- Definición de colores.
- Definición de iconos.

3.4.1 Variables de trayectoria

Para que el sistema sepa donde encontrar los diferentes elementos que debe buscar, se definen las siguientes variables:

- `#:ICEeditor:code-path`, que almacena el directorio en donde se encuentra el código LE-LISP del programa.
- `#:ICEeditor:help-path` almacena el directorio en el cual se encuentran guardados los ficheros de ayuda utilizados tanto por ICEeditor como por sus distintos componentes.
- `#:ICEeditor:icon-path` es la variables en la cual se almacena el directorio en el que se encuentran los iconos utilizados por ICEeditor.
- `#:system:icon-path` es la variable que utiliza el sistema AÏDA para determinar los directorios en los cuales buscar los iconos cuando se realizan llamadas a la función `libloadicon`. Su contenido es una lista de , cada una de las cuales contiene la ruta de un directorio. Al arrancar AÏDA dicha lista está formada por los siguientes directorios[3]:
  - `/usr/ilog/aida/icon/`
  - `/usr/ilog/aida/icon/color/`
  - `/usr/ilog/aida/icon/small/`
  - `/usr/ilog/aida/icon/large/`
  - `/usr/ilog/aida/icon/panel/`

  El orden en que aparecen estos directorios es importante ya que se toma el primer icono cuyo nombre coincida con el de un fichero con extensión `.i` en alguno de los directoios anteriores, buscando en el orden en que se han mostrado. Para incluir el contenido de `#:ICEeditor:icon-path` en los caminos de búsqueda de iconos del sistema, se utiliza la siguiente asignación:

```
(defvar #:system:icon-path (cons #:ICEeditor:icon-path #:system:icon-path))
```

3.4.2 Variables de mensajes multi-idioma

Para facilitar la interacción con el usuario se ha incorporado a ICEeditor la capacidad de mostrar todos los mensajes en varios de los idiomas más comunmente utilizados en el entorno en el cual se enmarca dicha aplicación. Concretamente, ICEeditor es capaz de utilizar indistintamente cualquiera de los siguientes idiomas:

- Inglés
- Francés
- Castellano
- Gallego

A menos que se indique otra cosa, ICEeditor comienza su ejecución utilizando el inglés como lengua de interacción con el usuario. Realmente todo el código de ICEeditor, incluyendo los comentarios, ha sido escrito en inglés, por imperativos de estandarización en el marco del proyecto en el cual se desarrolla este editor.

[3] Cuando se trabaja en AÏDA con nombres de directorios, éstos deben terminar en / (*slash*), puesto que la concatenación de nombres de directorios con nombres de ficheros se realiza de modo literal.

Los idiomas inglés y francés vienen ya predefinidos en LE-LISP, pero el castellano y el gallego ha sido preciso definirlos.

Los ficheros `spanish.ll` y `galician.ll` contienen la declaración del correspondiente idioma, la cual se realiza, en el caso del castellano, mediante la siguiente línea de código:

```
(record-language 'spanish)
```

Además, cada fichero contiene la traducción de todos los mensajes del sistema AÏDA, por lo que una vez arrancado ICEeditor, se podrá utilizar AÏDA también en estos dos nuevos idiomas.

Debido al elevado número de mensajes, puede llegar a agotarse el espacio de memoria reservado a las cadenas de caracteres, en cuyo caso puede obtarse por alguna de las siguientes soluciones:

1. Eliminar la carga de los ficheros `spanish.ll` y `galician.ll`, con lo cual tan sólo los mensajes internos de ICEeditor dispondrán de la capacidad de mostrarse en los dos idiomas adicionales.

2. Utilizar una versión reducida de los dos ficheros, dejando tan sólo aquellos mensajes globales del sistema que se consideren de interés para conseguir una mayor comodidad del usuario.

3. Incrementar el espacio de memoria que LE-LISP reserva a las cadenas de caracteres. Para ello es preciso recompilar el sistema LE-LISP, estableciendo la opción `string` de la variable `SIZE` del fichero `Makefile` al valor adecuado[4].

Los mensajes mostrados por ICEeditor que se han visto sujetos a traducción comprenden:

- Las opciones de los menús.

- Los botones que tienen asociado un texto.

- Los mensajes de ayuda.

- El texto de los diferentes tipos de cajas de diálogo que surgen como consecuencia de la comunicación entre el programa y el usuario.

Para cada uno de los mensajes ha sido definida una función que almacena las cadenas de caracteres con la equivalencia en cada idioma de una palabra dada.

Debido a problemas asociados a la implementación del método `redisplay`, el cual no adapta el tamaño de los objetos que contienen mensajes multi-idioma[5]. Se debe tener cuidado de que cada mensaje ocupe la misma porción de pantalla en todos los idiomas. No se trata de un problema trivial cuando se utilizan fuentes proporcionales. En este caso, si siempre se utiliza la misma fuente, se puede intentar ajustar manualmente.

[4] El valor de opción **`string`** se indica en múltiplos de 8 Kilobytes.

[5] Cuando, como resultado de un cambio de idioma, se va a utilizar un mensaje cuyo texto es mayor que en el idioma original, dicho texto aparecerá truncado en pantalla para adaptarlo al espacio ocupado por el original.

3.4.3 Variables para cursores

Cuando el usuario edita el texto de un componente léxico, la forma del cursor varía según sea el tipo de operación que se vaya a realizar. Para facilitar el mantenimiento del programa y mejorar la utilización de recursos se crean en la inicialización de ICEeditor todos los cursores que se van a utilizar, con lo cual se evita la creación de los cursores cada vez que se precisa un cambio. Con ello además se aumenta la velocidad de ejecución.

Concretamente, se han definido las siguientes variables:

- `#:ICEeditor:cursor-for-delete` que almacena el cursor utilizado cuando la operación a realizar es el borrado de un componente léxico. Por defecto, este cursor tiene la forma de una calabera.
- `#:ICEeditor:cursor-for-insert` que almacena el cursor utilizado en las operaciones de inserción. En principio, este cursor tinene la forma de un lápiz.
- `#:ICEeditor:cursor-for-modify`, el cursor de las operaciones de modificación. Su figura muestra dos flechas indicando un intercambio.
- `#:ICEeditor:cursor-for-info`, utilizado cuando el usuario desea obtener la información asociada a una componente léxico. Su forma coincide con la de un signo de interrogación.
- `#:ICEeditor:cursor-for-no-action`. Este cursor se utiliza cuando el texto aún no ha sido analizado, en cuyo caso el usuario dispone de libertad para realizar las modificaciones que estime comvenientes antes de proceder al análisis. La forma de este cursor es la de la típica barra de inserción utilizada por la mayoría de los editores.

Cada uno de los distintos cursores se carga utilizando la función `make-named-cursor`.

3.4.4 Variables de fuentes

ICEeditor hace un uso amplio de las fuentes, ya que tanto las opciones de los menús, el texto del editor como el título del fichero a editar aparecen en pantalla mostrados en diferentes fuentes. Cada fuente que va a ser utilizada en un elemento gráfico del sistema es almacenada en una variable. Para ello se han definido:

- `#:ICEeditor:texeditor-font`, la fuente utilizada en el editor de textos. Por defecto se corresponde con `kana` de tamaño 8×16[6].
- `#:ICEeditor:title-font`, que se utiliza para titular al propio ICEeditor.
- `#:ICEeditor:filename-font`, fuente utilizada para mostrar y editar el nombre del fichero que se está utilizando actualmente.
- `#:ICEeditor:button-font`, utilizada en aquellos botones que tienen una cadena de texto como imagen.
- `#:ICEeditor:menu-button-font`, que almacena la fuente utilizada para rotular los submenus.

[6] Las utilidades incluidas en el módulo **edps**[ILOG 91c] tienen problemas al tratar con este tipo de fuente. Por ello, en ciertos casos, se sustituye por **9x15**

- `#:ICEeditor:menu-options-font`, que indica la fuente en la que se mostrarán las opciones del menú.

Las fuentes se cargan utilizando la función `load-font`, a la que se le pasa como argumento la cadena de caracteres con el nombre completo de la fuente en formato X11[7].

Se deben tener muy presentes los problemas de compatibilidad. El conjunto de fuentes soportado difiere entre distintos servidores X. Para evitar problemas es conveniente emplear una de las dos opciones siguientes:

1. Utilizar los nombres abreviados. Por ejemplo, es mejor utilizar `8x16kana` o `9x15` como nombre de fuente que especificar el nombre X11 completo.
2. Rellenar con valores concretos sólo aquellos campos del nombre de la fuente que sean estrictamente necesarios. Por ejemplo, se podría específicar que se desea una fuente de cierta familia, con o sin inclinación y monoespaciada o proporcional, utilizando `*` en los demás campos[8].

Con ello se evita que la aplicación produzca un error al no estar disponible una fuente concreta. Como contrapartida no se garantiza que en todos los servidores X donde se ejecute la aplicación se muestre exactamente la misma fuente, aunque sí una similar.

3.4.5 Variables de colores

Los distintos colores utilizados en los componentes de ICEeditor se almacenan en variables. Concretamente, para cada componente se ha definido un par de variables, una de las cuales almacena el color del primer plano[9] y la otra el color del fondo[10]. Se han definido variables para los siguientes componentes:

- El editor de texto.
- El contorno del editor de texto.
- Las barras de desplazamiento.
- La barra de botones.
- El nombre del fichero que se está editando.
- El título de ICEeditor.
- El botón de insertar.
- El botón de modificar.
- El botón de eliminar.
- El botón de información.

Para crear un color se utiliza la función `make-named-cursor` a la que se pasa como parámetro la cadena de caracteres que identifica al color requerido en el sistema X11.

[7] X window System versión 11.

[8] Algunas implementaciones del X Window System proporcionan una aplicación denominada **`xfontsel`** que facilita enormemente la tarea de selección de fuentes.

[9] Foreground.

[10] Background.

3.4.6 Variables de iconos

Ciertos iconos utilizados en las ventanas de diálogo y que por lo tanto no están siempre presentes en pantalla, son almacenados en variables. Aquellos iconos que siempre permanecen visibles, concretamente los icorporados en la barra de botones, se llaman directamente en la función de creación de ICEeditor. Reservar espacio por medio de variables para su almacenamiento no reporta ningún beneficio en tiempo de ejecución ni en ahorro de espacio.

Las variables utilizadas para almacenar iconos son:

- `#:ICEeditor:ok`, que contiene el icono utilizado en el botón de confirmación que aparece en las ventanas de diálogo.
- `#:ICEeditor:cancel`, idem para el botón de cancelación.
- `#:ICEeditor:wait` almacena el icono utilizado en la ventana de presentación inicial cuando el usuario debe esperar a que se realice la carga del módulo correspondiente al `textedit`.
- `#:ICEeditor:warn-icon` se utiliza para almacenar el icono que aparece en las ventanas de advertencia.
- `#:ICEeditor:confirm-icon` se utiliza para guardar el icono que identifica a las ventanas de confirmación.
- `#:ICEeditor:info-icon` es utilizado en la ventana donde se le muestra al usuario la información relativa a un componente léxico.

El proceso de carga de un icono involucra los siguientes pasos:

1. Comprobar que el icono no ha sido ya cargado. Con ello evitamos el tiempo de procesamiento requerido por la carga en caso de que ésta no sea necesaria y se mejora el rendimiento general del sistema al evitar cargar nuevamente en memoria datos ya disponibles[11].
2. En caso de que el icono no exista, cargarlo utilizando la función `libloadicon`. A esta función se le pasa como argumento un átomo, no una cadena de caracteres. El nombre de este átomo debe coincidir con el del fichero, sin la extensión `.i`, en el cual se encuentra almacenado el bitmap que se va a utilizar para crear la imagen del icono.
3. En caso de haber sido cargado previamente, utilizar la copia existente en memoria mediante la asignación a la variable de ICEeditor de la variable correspondiente al icono.
4. Utilizar la función `icon` o `maskicon` para crear el icono con el valor devuelto en cualquiera de los pasos 2 ó 3.

[11] AÏDA crea una variable interna por cada icono que se carga, cuyo nombre coincide con el del icono cargado. Utilizando la función **boundp** se puede conocer si dicha variable existe (y por consiguiente el icono).

3.5 Jerarquía de componentes

Como ya se ha dicho anteriormente, ICEeditor es una aplicación compleja que está formada por un conjunto de subaplicaciones, la mayor parte de ellas objetos AÏDA estándar con el comportamiento redefinido, esto es, se han utilizado métodos propios para establecer las acciones que se realizan cuando tiene lugar un evento asociado a la subaplicación.

La jerarquía de componentes desempeña un papel fundamental en el funcionamiento de la aplicación, puesto que establece un modo fácil de comunicación entre los distintos componentes. Así, por ejemplo, es posible llamar a un método de la aplicación principal ICEeditor cuando el usuario dispara la acción asociada a la pulsación de un botón. El mecanismo utilizado por LE-LISP para la gestión de componentes esconde gran parte de estos problemas de interacción de la aplicación y facilita el problema siempre latente de mantenimiento del software, puesto que es posible añadir un nuevo componente sin afectar el normal funcionamiento de los demás.

En ICEeditor se ha considerado conveniente definir las siguientes subaplicaciones componentes[12]:

- `ICEEDITOR` es el componente asociado a la aplicación ICEeditor. La creación de este componente es necesaria para permitir a las subaplicaciones acceder a la instancia de la aplicación `ICEeditor` a la que pertenecen. También posibilita el acceso a los métodos definidos para `ICEeditor` desde las acciones asociadas a las subaplicaciones.
- `TEXTEDITOR`, el componente que se refiere el editor de textos incorporado en ICEeditor.
- `FILENAMEEDIT`, el editor de cadenas de caracteres que permite al usuario cambiar interactivamente el nombre del fichero que está editando.
- `MAINMENU` se corresponde con el menú principal de la aplicación.
- `SCROLLER` se refiere al desplazador vertical encargado de desplazar las líneas de texto del editor siguiendo los movimientos del cursor.
- `SCROLLBAR` se refiere a la barra de desplazamiento horizontal asociada al editor de textos.
- Para cada uno de los botones se ha definido un componente individual:
  - `QUITBUTTON` para el botón asociado a la acción de salir de ICEeditor.
  - `PARSERBUTTON` para el botón de parsing incremental.
  - `PARSERALLBUTTON` para el botón de reanálisis completo.
  - `LOADBUTTON` para el boton de carga de ficheros.
  - `SAVEBUTTON` para el botón de escritura de ficheros.
  - `INSERTBUTTON` para el botón que pone al editor en modo inserción.
  - `DELETEBUTTON` para el botón de la acción de borrado de componentes léxicos.

[12]Los componentes se han nombrado en mayúsculas para evitar confusiones en los casos en que el componente recibe la misma denominación que otra variable de ICEeditor.

- `MODIFYBUTTON` para el botón de modificación.
- `NOACTIONBUTTON` para el botón que inhibe las operaciones de edición sobre los componentes léxicos en el editor.
- `INFOBUTTON` para el botón que permite al usuario obtener la información de análisis almacenada para cada componente léxico.
- `HELPBUTTON` para el botón de acceso a la ayuda.

Para incorporar cada uno de los elementos anteriores a la jerarquía de componentes se utiliza la función `ADD-COMPONENT`. Por ejemplo, en el caso del botón quit:

```
(add-component ICEeditor 'QUITBUTTON quitbutton)
```

Con ello se consigue establecer que `quitbutton`, una variable creada previamente con su valor acotado dentro del ámbito de un `let`, se incorpore a la jerarquía de componentes de la variable `ICEeditor` de tipo {`ICEeditor`} que también había sido creada dentro del ámbito del mismo `let`.

3.6 Construcción de la imagen

En AÏDA, todo objeto descendiente de la clase {`application`} posee un campo predefinido cuyo nombre es `image` en el cual se almacena la imagen gráfica del objeto, que deberá ser del tipo {`image`}. Puesto que la clase {`application`} desciende también de la clase {`image`}, se pueden incluir aplicaciones en el campo `image`.

Existe un método asociado a ICEeditor, cuyo nombre nombre es `create-image`, que se encarga de crear una nueva imagen para cada instancia de {`ICEeditor`} que se crea. Este método es invocado por la función `create-ICEeditor` de la siguiente forma:

```
(send 'new-image ICEeditor ({ICEeditor}:create-image ICEeditor))
```

El método `new-image` permite establecer o sustituir la imagen existente de un objeto de tipo aplicación, es decir, establece el valor del campo `image` para una aplicación.

En el método `create-image` se definen una serie de variables locales que almacenan el tamaño horizontal y vertical de la imagen de los diversos componentes de la imagen. Con ello se evita tener que realizar constantemente llamadas a los métodos que retornan las dimensiones de los objetos, puesto que son valores ampliamente utilizados para lograr que la imagen se muestre adecuadamente en pantalla.

3.6.1 Construcción del menú

El menú constituye uno de los elementos de interacción de los que dispone el usuario para comunicarse con la aplicación. AÏDA proporciona funciones que facilitan las tareas involucradas en la creación de menús, aunque no por ello deja de ser una tarea pesada en cierto grado.

Desde la función encargada de la creación de la imagen se llama al método `create-mainmenu`, que como su nombre indica, es el encargado de crear el menú que va a formar parte de la imagen de ICEeditor.

El menú de ICEeditor va a contar, desde un punto de vista meramente estructural, de las siguientes partes:

- Una barra de menú en la cual aparecerán los títulos genéricos que engloban los grupos de opciones definidos en los submenús.
- Un menú desplegable por cada una de las opciones indicadas en la barra de menú.
- En ciertos casos más menús desplegables de inferior nivel asociados a opciones presentes en menús desplegables del nivel inmediatamente superior.
- Un botón asociado a cada una de las opciones de un menú de cualquier nivel. Las opciones cuya pulsación desencadena la aparición de un submenú son botones del tipo `menubutton`.

La barra del menú principal

Para crear la barra de menú se utiliza la función `create-aidamenubar`, a la que se pasan como argumentos los títulos de cada uno de los botones que la componen, esto es, el contenido del campo imagen de cada uno de ellos. En el caso de ICEeditor, se utilizarán cadenas de caracteres para rotular dichos botones, pero AÏDA no plantea restricciones al respecto.

Como nuestra intención es asociar submenús a las opciones de la barra, no definimos ninguna acción específica para los botones que la componen. Posteriormente, una vez se hallan creado los submenús, se utilizará la función `aidamenu-attach-submenu` para crear el vínculo que asocie cada botón con el menú desplegable correspondiente.

Los menús desplegables

Cada uno de los menús de tipo desplegable que componen el menú principal se define utilizando la función `create-aidamenu`. Su funcionamiento es similar al de la función utilizada para crear la barra de menú.

En la mayoría de las funciones se utiliza la función `messagestring` para acceder a la cadena de caracteres del idioma actual contenida en alguno de los mensajes multi-idioma previamente definidos. En algunos casos se utiliza una combinación de iconos y texto para rotular las opciones de menú. En tales casos, se utiliza el constructor de imágenes `row` para construir la imagen del botón correspondiente.

Si se desean establecer varios niveles de menús desplegables, se debe pasar el valor `()` al parámetro encargado de asociar acciones a las opciones de menú y utilizar posteriormente la función `aidamenu-attach-submenu` para enlazar los menús desplegables entre sí.

Si se desea que la selección de una opción dispare una acción, entonces de debe pasar como uno de los parámetros de la función de creación del menú la función que se disparará en el caso de que se lleve a cabo la selección. Hay dos maneras de hacer esto:

1. Pasar el nombre de una función previamente definida.
2. Definir *in situ* una función anónima[13].

En cualquier caso, se debe tener en cuenta que la función utilizada, sea anónima o con nombre, debe tomar como único argumento el menú desde el cual se disparó dicha acción.

[13] LE-LISP permite definir funciones anónimas mediante el uso de la forma `lambda`

En ICEeditor, se ha obtado mayoritariamente por utilizar funciones anónimas desde las cuales se invocan métodos de {`ICEeditor`}, al que se accede por medio de la jerarquía de componentes. La causa principal de esta elección se debe a la abligatoriedad de pasar el menú como argumento, ya que así evitamos la dependencia del orden de las opciones en el menú.

3.6.2 Construcción de la barra de botones

La barra de botones está constituida por un conjunto de botones situados en el lado derecho del editor de textos. Ha sido pensada como una ayuda al usuario con el fin de que éste sea capaz de acceder cómodamente a aquellas funciones de uso más frecuente.

Los botones que componen la barra se crean en la función `create-ICEeditor` mediante llamadas a la función `standardbutton`. La imagen de cada botón está formada por un icono de tipo {`elasticmaskicon`} cuyo bitmap se encuentra almacenado en disco.

Una vez que han sido creados los botones, se procede a su inclusión en la jerarquía de componentes mediante llamadas a la función `add-component`.

Después se asocia a cada uno de ellos la acción que se disparará cuando el usuario pulse el ratón dentro del área de la pantalla ocupada por la imagen del botón.

Las funciones que definen el comportamiento de un botón deben de tener como único argumento la instancia del botón en el cual han sido disparadas. En ICEeditor se ha obtado por la utilización de funciones anónimas desde las cuales se invocan a métodos de ICEeditor. Con ello se ha pretendido evitar la definición de multitud de funciones con nombre cuya única misión sería la de actuar de intermediarias entre el botón y el método de la clase {`ICEeditor`}.

3.6.3 Contrucción del editor

Sin duda, el elemento más importante de la imagen de ICEeditor es el editor de textos que incorpora. Esta importancia viene dada por las siguientes causas:

- Es la visión que se le muestra al usuario del editor de componentes léxicos.
- Está íntimimante relacionado con las estructuras de datos que dan soporte a la estrategia de asociación entre componentes léxicos y el texto que se haya utilizado.
- Realiza las funciones de interfaz entre el usuario y los componentes léxicos.
- Es el principal elemento de interacción del usuario con ICEeditor.
- Los demás componentes de la imagen tienen como fín establecer las acciones que se pueden realizar sobre el editor así como facilitar su realización en la medida de lo posible.

En la función `create-ICEeditor` se crea el componente `TEXTEDITOR` mediante una llamada a la función `create-texteditor-of-ICE`.

Los objetos de tipo {`textedit`} no están incorporados en la jerarquía estándar de clases de AÏDA, sino que es preciso cargar el módulo `textedit` para tener acceso a este tipo de editores. La carga de un módulo consume bastante tiempo: dependiendo del tamaño del módulo se puede tardar incluso del orden de una decena de segundos. Una

vez que un módulo ha sido incorporado al sistema, no es preciso volver a cargarlo durante el resto de la sesión. Es conveniente realizar una llamada a la función `featurep` antes de intentar cargar un módulo. Con ello se evitará perder tiempo en tareas inecesarias y se realizarán menos llamadas a las rutinas del *garbage collector*.

La expresión `(featurep 'textedit)` devuelve `()` si el módulo `textedit` aún no ha sido cargado. En tal caso se procederá a cargarlo mediante la función `loadmodule`.

Características del editor

Los editores `textedit` son lo únicos editores de AÏDA que se adaptan a los requerimientos de ICEeditor:

- Permiten mostrar cualquier tipo de imagen AÏDA, no sólo cadenas de caracteres.
- Permiten establecer individualmente los atributos de cada uno de los caracteres (o imágenes). En lo que a ICEeditor concierne, esto es importante, ya que con ello es posible utilizar difererentes colores y fuentes para cada operación de edición de componentes léxicos.
- Actúa correctamente con cualquier tipo de fuente, no sólo con las monoespaciadas. Con ello se amplía la gama de fuentes entre las que poder optar.

Sin embargo, también presentan cietos inconvenientes, entre los cuales detacan:

- No adaptan su tamaño a la longitud de las líneas del texto que se muestra.
- No existe correspondencia entre las línea en el texto[14] y las líneas del editor[15]. Si una línea es demasiado grande para caber en una línea del editor, entonces el resto de la línea de texto es pasado a la siguiente línea del editor.
- Cambios en el tamaño del editor pueden provocar cambios en la distribución de las líneas:
  - Si el tamaño del editor aumenta horizontalmente, una línea de texto que antes era distribuida en varias líneas del editor, ahora puede caber en una sola.
  - Si la anchura del editor disminuye, una línea de texto que antes cabían completamente en una línea del editor puede que ahora tenga que ser repartida entre varias.
- No tienen capacidad de desplazamiento horizontal cuando el cursor sobrepasa los límites visibles.

En ICEeditor es preciso mantener en todo momento la correspondencia entre los componentes léxicos y el texto que los representa en el editor. Las modificaciones de la distribución de las líneas en el editor trastoca esta correspondencia, puesto que no se puede utilizar el concepto de línea como elemento fiable.

Una posible solución consiste en impedir las operaciones de redimensionamiento del editor. Esta no es una opción aceptable por las siguientes razones:

[14] Secuencia de caracteres terminadas con un carácter de fin de línea, típicamente un carácter de *linefeed* en los sistema Unix.

[15] Una línea del editor está formada por la secuencia de caracteres o imágenes que caben en la anchura actual del editor.

- Limita en gran medida la capacidad del usuario de adaptar ICEeditor a sus preferencias.
- Es difícil predecir de antemano cual es el tamaño idóneo en cada sesión, puesto que se pueden editar distintos fichero, con textos de distintas características.

La solución adoptada finalmente en ICEeditor puede considerarse un punto intermedio entre la anarquía de líneas de la versión original y la rigidez de la ausencia de redimensionamiento:

- El componente `TEXTEDITOR` tiene internamente una anchura fija.
- La imagen mostrada al usuario tiene tamaño variable. El usuario puede modificar sus dimensiones según sus preferencias.

El editor de textos se crea con una anchura suficiente como para dar cabida a prácticamente cualquier línea. Se ha optado por una anchura equivalente a 255 caracteres. No se permiten líneas de mayor longitud. Si alguna supera estas dimensiones, será truncada.

El editor se encapsula en un desplazador cuya anchura es inicialmente de 80 caracteres. Cuando el usuario modifica el tamaño de la ventana de ICEeditor, el elemento que varía de tamaño es el desplazador, no el editor.

Inicialización del editor

La función `textedit` devuelve un editor de tipo {`textedit`}. Al llamarla se le deben pasar cuatro parámetros: las coordenadas vertical y horizontal, la anchura y la altura. Para calcular las dimensionas iniciales se utilizan las macros `#<valor>hchar` y `#<valor>wchar` que devuelven, respectivamente, el valor en pixels de la anchura y la altura de un caracter en la fuente utilizada cuando fueron llamadas. Se puede cambiar temporalmente la fuente utilizada encerrando la llamada a `current-font` dentro del ámbito establecido por un `with`.

Mediante una llamada a `te-keep-redisplay-regions` se pone a `t` el *flag* que establece cómo se realizan las actualizaciones de la imagen cuando hay cambios en el editor. Con el *flag* a `t`, se va almacenando una historia de las modificaciones realizadas. Con ello es posible realizar operaciones complejas sin necesidad de redibujar el editor al final de cada una de ellas, sino que se van acumulando y al finalizar todas ellas es cuando tiene lugar el `redisplay` de la imagen[16].

Como se tendrán que capturar los eventos provocados por la pulsación del botón izquierdo del ratón sobre el editor, se utiliza la variable `#:ICEeditor:texteditor:down0` para almacenar la función estándar utiliza por el editor cuando ocurre tal tipo de eventos en condiciones normales de operación. Para ello se utiliza el código del siguiente fragmento de programa:

```
(defvar #:ICEeditor:texteditor-down0 'te-mouse-generate-display-selection)
```

[16] Este tipo de *redisplay retardado* sólo es posible cuando se utiliza el editor en modo programado, esto es, cuando se utilizan funciones LE-LISP para modificar los contenidos del editor. Las modificaciones interactivas, aquellas realizadas directamente por el usuario haciendo uso del ratón y del teclado, provocan una actualización inmediata de la imagen.

La definición de estilos

Como se mencionó anteriormente, es posible utilizar múltiples fuentes y colores en un editor de tipo `textedit`. Un *estilo* está compuesto de cuatro elementos:

- Una fuente.
- Un *flag* que indica si se utiliza subrayado.
- Un color para el primer plano.
- Un color para el fondo.

Para que un conjunto de caracteres aparezcan en pantalla en un determinado tipo de fuente y color, hay que realizar los siguientes pasos:

1. Crear un nuevo estilo mediante `te-define-style` con la fuente y los colores deseados.
2. Aplicar el estilo al grupo de caracteres seleccionados.

En ICEeditor se utilizan los siguientes estilos:

- `te-main-style`. Este es un estilo predefinido en todos los `textedit` que establece las características generales del texto.
- `#:ICEeditor:te-insert-style`, que se utiliza para resaltar los caracteres introducidos en el transcurso de una inserción.
- `#:ICEeditor:te-delete-modify`, utilizado para resaltar los caracteres pertenecientes a componentes léxicos sobre los que se ha aplicado la operación de borrado.
- `#:ICEeditor:te-modify-style`. Este estilo se aplica a los caracteres que pertenecen a componente léxico sobre los que ha tenido lugar al menos una operación de modificación.
- `#:ICEeditor:te-info-style` se utiliza para indicar los caracteres que pertenecen al componente léxico sobre el cual se está solicitando información.

3.6.4 La definición de selecciones

Para aplicar un estilo a un conjunto de caracteres es preciso que dichos caracteres estén incluidos en alguna *selección*. Una selección no es más que un conjunto de zonas del texto. Las zonas que componen una selección no tienen porque ser contiguas, sino que pueden estar repartidas por todo el texto. El tamaño de las selecciones es dinámico, con lo cual es posible añadir y eliminar caracteres de una selección.

Cuando se crea una selección mediante `te-create-selection` se le asigna un estilo. Todos los caracteres incluidos dentro de una selección se mostrarán en pantalla utilizando el estilo asociado a dicha selección.

En ICEeditor se utilizan cuatro selecciones, una para cada tipo de operación:

- `#:ICEeditor:te-insert-sel`, en la que se van incluyendo los caracteres de los diferentes componentes léxicos insertados. Está asociada el estilo `#:ICEeditor:te-insert-style`.

- `#:ICEeditor:te-delete-sel`. Todos los caracteres de los componentes léxicos borrados se incluyen en esta selección, cuyo estilo asociado es `#:ICEeditor:te-delete-style`.

- `#:ICEeditor:te-modify-sel` incluye los caracteres de los componentes léxicos que han sido objeto de una operación de modificación. Se incluyen tanto los caracteres originales que no hayan sido borrados como los nuevos caracteres insertados. El estilo que se utiliza es `#:ICEeditor:te-modify-style`.

- `#:ICEeditor:te-info-sel`, que muestra con las características del estilo `#:ICEeditor:te-info-style` los caracteres del componente léxico sobre el que se ha solicitado información.

Los cursores

Un cursor es un punto localizado entre dos caracteres del texto que indica el lugar en el que tendrán efecto las modificaciones realizadas. No se deben confundir los cursores del editor `textedit` con los cursores del VBP[17] de LE-LISP[18]. Existe un cursor predefinido denominado `insertion-cursor` cuya posición se corresponde con la del icono utilizado para mostrar al usuario el lugar en el que está insertando texto, borrando texto o simplemente el lugar en donde se ha situado utilizando la teclas de movimiento del cursor o el ratón[19]. Es por tanto el cursor utilizado para las interacciones mediante el teclado.

Todas las funciones que utilizan un cursor como parámetro opcional asumen el `insertion-cursor` como valor por defecto.

En ICEeditor se hace uso de los cursores para determinar el alcance de cada operación de edición. Para ello se definen los siguientes cursores:

- `#:ICEeditor:te-char-cursor` se utiliza como cursor auxiliar.

- `#:ICEeditor:te-begin-op-cursor` se utiliza para determinar el primer carácter involucrado en una operación de edición.

- `#:ICEeditor:te-end-op-cursor` determina la posición del último carácter implicado en la edición de un componente léxico.

3.6.5 Construcción del desplazador del editor

El usuario puede moverse libremente por el texto utilizando el cursor, pero en ciertos casos en que se desea dar grandes saltos en el texto, es conveniente disponer de elementos como las barras de desplazamiento. La misión de estas barras es permitir *deslizar* el texto tanto horizontal como verticalmente sin necesidad de pulsar repetidamente las teclas de movimiento del cursor. Accionando con el ratón las barras de scroll es posible realizar rápidos desplazamientos por todo el texto.

[17] **V**irtual **B**itmap **D**isplay.
[18] Los cursores del VBD se refieren a los bitmaps utilizados para mostrar el puntero del ratón o del cursor de texto.
[19] Por tanto habrá un cursor del VBP situado en la posición indicada por el `insertion-cursor`.

La barra de scroll vertical

Los editores {`textedit`} admiten la posibilidad de ser enlazados con una aplicación de tipo {`scrollbar`}. Dicho enlace se realiza mediante la función `te-attach-scrollbar`. Durante la ejecución de la función `create-ICEeditor` se crea una barra de desplazamiento vertical mediante la función `verticalsrollbar`. Dicha barra se integra en la jerarquía de componentes de ICEeditor, recibiendo el nombre de `SCROLLBAR`.

Una vez situada la barra de desplazamiento en la imagen, hecho que ocurre durante la ejecución del método `create-image`, se establece una perfecta interacción entre ella y el editor, de modo que que cualquier cambio de posición en uno de los dos elementos se ve inmediatamente reflejado en el otro.

El desplazador horizontal

Aunque resulte paradógico, los objetos de tipo {`textedit`}, que tan hábilmente interactúan con la barra de scroll vertical, no proporcionan ningún medio directo de enlace con una barra de scroll horizontal.

Otros tipos de editores, como por ejemplo {`medite`}, utilizan una correspondencia biyectiva entre líneas de texto y líneas de editor. En ellos el texto se desplaza en la dirección correcta cuando el cursor alcanza el límite derecho o izquierdo de la ventana del editor.

Los editores `textedit` desplazan el cursor a la línea siguiente cuando se alcanza el borde derecho y a la línea precedente cuando se alcanza el borde izquierdo.

Como ya se comentó anteriormente, esta es una de las causas que provocaron la elección de un tamaño fijo lo suficiantemente grande para permitir la edición de la mayoría de los textos. Como este editor completo no cabría en la pantalla, es preciso utilizar un *desplazador* que acote la imagen del editor y permita al usuario desplazarse por toda su superficie mediante el uso del ratón.

El editor ignora la existencia del desplazador, por lo que no le advierte en ningún momento de cuándo se debe realizar un desplazamiento provocado por el movimiento del cursor de inserción. Es preciso capturar los eventos asociados al teclado para actualizar manualmente la porción del editor que deberá estar visible en la ventana del desplazador.

El desplazador se incorpora a la jerarquía de componentes con el nombre de `SCROLLER`.

Combinación de los desplazadores

De lo comentado en los puntos anteriores se desprende que las barras de desplazmiento horizontal y vertical son totalmente independiente, estando ligadas al editor por medios también enteramente distintos:

- En el caso de la barra vertical, es el propio editor el que se encarga de mantener la coordinación mediante la interacción directa con la barra de scroll.
- En el caso de la barra horizontal, se tienen dos características diferenciadoras:
  - No es una aplicación independiente como la vertical, sino que forma parte de un desplazador.

– La interacción entre este desplazador y el editor se ha programado totalmente y se realiza mediante la captura de los eventos del teclado.

Sin embargo, al construir la imagen, se han situado de tal modo que el usuario tiene la sensación de que están integradas en un único desplazador. Con ello se demuestra una vez más la importancia de separar lo que es el comportamiento de un objeto de lo que es la imagen en sí.

3.6.6 Combinación de todos los elementos

Una vez que se han creado todos los componentes de la imagen surge el problema de su disposición. Se deben de considerar los siguientes aspectos:

1. La posición relativa que ocupará cada componente con respecto a los demás. Se obtendrá un esquema en el que se indicará cosas como por ejemplo que la barra de botones estará situada a la derecha del editor o que las barras de scroll estarán a la izquierda y abajo en el editor. Con ello se obtendrá una configuración general de la imagen.

2. Las dimensiones iniciales de la imagen y la proporción ocupada por cada componente en ella.

3. Cómo reaccionará cada componente ante los eventos de redimensionamiento que tengan lugar en la ventana de la aplicación.

La solución adoptada en lo referente a los dos primeros puntos depende en gran medida de la estética del diseñador. Se debe tener siempre en mente la importancia del confort del usuario. Puede ser preciso sacrificar la estética en aras de conseguir una mayor ergonomía y facilidad de uso.

El tercer punto está más ligado al diseño y programación de la aplicación. El tamaño de la ventana utilizada por ICEeditor puede verse modificado como consecuencia de la acción del usuario, por razones variadas:

- Para incrementar la porción de texto visible en el editor.

- Para acomodar el tamaño de ICEeditor al espacio dejado por las demás ventanas abiertas de modo que se pueda trabajar cómodamente en todas ellas.

- Por preferencias personales.

Ciertamente si se modifica el tamaño de la ventana se ha de producir una modificación el el tamaño de la imagen. Los distintos componente se repartirán el espacio manteniendo las mismas posiciones relativas y a ser posible las mismas proporciones con respecto a la imagen total. En una aplicación compleja como ICEeditor se pueden establecer distintas clases de componentes si tomamos como base su respuesta al cambio de tamaño:

- Componentes que no varían su tamaño. Se da en el caso de los menús y de los botones. Este tipo de compotamiento de debe tanto a consideraciones estrictamente

técnicas (los iconos y las cadenas de caracteres no pueden variar su tamaño[20]) como prácticas[21] y estéticas[22].

- Componentes que adaptan sólo una de sus dimensiones al nuevo tamaño de la ventana. Esto ocurre en el caso del editor de cadenas de caracteres utilizado para editar el título del fichero. Sólo modifica su anchura cuando se modifica la anchura de la ventana. Su altura permanece siempre invariable.
- Componentes que modifican su tamaño en sintonía con la modificación del tamaño de la ventana. Este es el caso del editor de textos.

En AÏDA existe el constructor de imágenes `constrainedview`. Este constructor es una variación de `view`, la superposición de imágenes. A diferencia de éstas, donde todos los objetos que forman la imagen crecen proporcionalmente, una `constrainedview` permite establecer *puntos de anclaje* para dichos objetos. Los anclajes se realizan con respecto a los bordes de la ventana.

Cuando un objeto tiene anclado uno de sus bordes, se mantendrá la distancia entre el borde del objeto y el borde de la ventana. La única excepción viene dada por el establecimiento de un tamaño mínimo para la aplicación. Dicho mínimo viene dado por el tamaño inicial establecido cuando se creó la aplicación.

Mediante a función `constrainedview` se crea un objeto imagen de tipo {`constrainedview`}. Esta función toma como parámetros los objetos que forman la imagen. Posteriomente se utiliza el método `update-constraint` para establecer los puntos de anclaje de cada uno de dichos objetos. Los puntos de anclaje se almacenan en objetos {`strechconstraint`}, que contiene el anclaje para los cuatro bordes de la *bounding-box* del objeto.

3.7 El comportamiento de ICEeditor

El comportamiento de cualquier objeto viene definido por sus métodos. En ICEeditor los métodos se *disparan*, esto es, se ejecutan, en respuesta a la ocurrencia de los siguientes eventos:

- La selección de una opción de menú.
- La pulsación de uno de los botones.
- La pulsación del botón izquierdo del ratón en el texto del editor.
- La pulsación de una tecla cuando el focus del teclado se encuentra en el editor.

3.7.1 Métodos de gestión de ficheros

Se activan en respuesta a la selección de una de las opciones del menú de *archive*. En este apartado se incluyen los siguientes métodos:

[20] Se puede modificar el tamaño de una cadena de caracteres utilizando un tamaño de fuente diferente. Sin embargo ciertos tipos de fuentes son difíciles de escalar automáticamente para cualquier tamaño, además de los problemas surgidos como consecuencia de la compatibilidad de fuentes entre servidores X.

[21] Unos iconos y menús demasiado pequeños o demasiado grandes resultan incómodos de manejar.

[22] Unos botones de tamaño excesivamente grande o pequeño provocan un cierto rechazo por su apariencia visual.

- `load-file`. Se activa tanto por la selección de la apción *load* del menú como por la pulsación del botón de *load* presente en la barra de botones. Su finalidad es la de cargar un nuevo fichero en el editor, reseteando la información de ICEeditor. Verifica la consistencia del fichero anterior con la versión almacenada en disco. Si no son iguales, se avisa al usuario para que éste decida si realmente se realiza la carga del nuevo fichero despreciando los cambios en el anterior.

- `new-file`. Está asociado a la opción *new* del menú. Su actuación es similar a la de `load-file` excepto que en vez de cargar el texto de un fichero del disco, inicializa ICEeditor con el nombre de fichero por defecto y sin texto en el editor.

- `revert-file`. Activado mediante la selección de la opción *revert* del menú. Conceptualmente es equivalente a `load-file` utilizando la versión almacenada en disco del fichero actualmente editado como fichero a cargar.

- `save-as-file`. Está asociado a la opción de menú *save-as*. Activa una ventana de diálogo para solicitar un nuevo nombre al fichero que se está editando procediendo a continuación a grabar su contenido en disco.

- `save-file`. Se activa al seleccionar la opción *save* del menú. También se activa mediante la pulsación del botón *save* de la barra de botones. Salva en disco el contenido del texto localizado en el editor.

- `quit`. Se puede activar tanto desde la opción *quit* del menú como pulsando el botón *quit* de la barra de botones. Su misión es la de terminar la sesión de ICEeditor. Libera todos los recursos previamente utilizados por la aplicación. Comprueba si se ha modificado el contenido del editor desde la última grabación. En caso afirmativo, requiere confirmación del usuario.

3.7.2 Métodos que activan la edición de componentes léxicos

En respuesta a la selección de alguna de las opciones presentes en el menú *edit* o a la pulsación de los botones de *insert*, *delete*, *modify no-action* o *info*, se dispara el método `change-edit-state`.

Si el texto no ha sido previamente analizado no es posible editar los componentes léxicos, puesto que no está disponible la estructura de asociación entre los componentes léxicos y el texto. En tal caso se envía un mensaje al usuario y termina la ejecución del método.

Si el texto está analizado, este método realiza las siguientes acciones:

1. Termina la operación de edición precedente. El comienzo de una nueva operación de edición constituye un punto de sincronismo en el cual debe hacerse consistente la información almacenada en el campo `TTLT`[23] y el texto almacenado en el editor.

2. Cambiar la forma del cursor. En cada operación de edición se utiliza un cursor diferente para facilitar la actuación del usuario. Para ello se llama a la función `change-cursor`, la cual establece el nuevo cursor restringiendo su ámbito de aplicación a la ventana del editor.

[23] **T**oken **T**ext **L**ink **T**able.

3. Invalidar el valor de los cursores `#:ICEeditor:te-begin-op-cursor` y `#:ICEeditor:te-end-op-cursor`. Esto es necesario para evitar malas interpretaciones de su valor en otras funciones de ICEeditor.

4. Cambiar el valor del campo `edit-state` al de la acción seleccionada.

5. Si la acción seleccionada es `no-action` o `info`, se establece el editor en modo de sólo lectura para evitar que su contenido sea modificado.

3.7.3 Métodos de personalización

Al seleccionar alguna de las opciones del menú titulado *options* se disparan los métodos de personalización de entorno:

- El método `change-language` establece el idioma que se va a utilizar en los elementos de la interfaz. Para ello se realiza una llamada a la función `current-language`. Se utiliza `full-redisplay` para cambiar inmediatamente todos los mensajes de ICEeditor.

- La macro `change-color` se llama cuando se selecciona alguna de las opciones para cambiar el color del texto afectado por las operaciones de edición. Esta macro realiza las siguientes acciones:

  1. Activa una ventana de diálogo para introducir el nuevo color.
  2. Redefine los estilos asociados al editor de texto y los reasigna a las correspondientes selecciones.
  3. Vuelve a crear el componente `MAINMENU` de la imagen. Este paso es necesario puesto que el submenú de colores muestra las distintas opciones de color utilizando los colores actuales.
  4. Recrea la imagen llamando al método `create-image`. Puesto que esta función no crea los componentes de la imagen, sino que tan sólo accede a ellos mediante `component` para crear la `constrainedview`, el resultado es una nueva imagen con los mismos componentes inalterados, excepto el menú principal, que ha sido sustituido por el nuevo generado en el paso enterior.

3.7.4 Métodos de llamada al parser

La interacción entre ICEeditor y el analizador sintáctico se realiza mediante los dos métodos siguientes:

- `parser-all` es el método utilizado para realizar un análisis completo del todo el texto almacenado en el editor. Independientemente de si el texto había sido analizado previamente o no, cuando se ejecuta este método se considera todo el texto como no analizado, reinicializando la información almacenada en los campos de {`ICEeditor`}, por lo que se elimina cualquier asociación entre los componentes léxicos y el texto creada previamente. Se puede acceder a este método pulsando el botón *parser-all* o seleccionando la opción *parser-all* del menú.

- `parser` es el método utilizado para realizar un análisis incremental del texto. Para ello se informa a ICE de las variaciones introducidas en los componentes léxicos y se actualiza conveniente la TTLT. Una llamada a este método sobre un texto que no ha sido nunca analizado provoca una llamada al método `parser-all`. Este método puede ser activado por el usuario mediante la pulsación del botón *parser* o mediante la selección de la opción *parser* en el menú.

3.8 El comportamiento del editor de componentes léxicos

Una parte muy importante del funcionamiento de la aplicación ICEeditor viene determinada por el comportamiento de uno sus componentes: el editor de componentes léxicos. Realmente, una parte considerable del código de ICEeditor está dedicada a la definición de funciones que se encargan de transformar las acciones del usuario sobre el editor en las correspondientes operaciones sobre las estructuras de asociación entre componentes léxicos y texto.

En la sección anterior se trató el tema del comportamiento general de ICEeditor en respuesta a los eventos producidos por el usuario que modificaban el entorno general de trabajo o la apariencia externa de la interfaz.

En esta sección se va a estudiar el comportamiento de ICEeditor ante las acciones directamente involucradas en la edición y manipulación de los componentes léxicos. Puesto que el editor de componentes léxicos es el componente construido más directamente sobre las estructuras de almacenamiento de los componentes léxicos, es lógico que sean los eventos producidos en el editor los que actúen sobre dichas estructuras de datos.

3.8.1 Activación del editor de componentes léxicos

En la interacción del usuario con el editor se pueden distinguir los dos casos siguientes:

- Cuando se ha cargado un texto y éste aún no ha sido analizado, el usuario puede modificar libremente el texto para proceder cuando estime conveniente a realizar un análisis léxico-sintáctico. En este caso la intervención de las funciones propias de ICEeditor es escasa, confiando en los métodos predefinidos para las instancias de `textedit`, los cuales permiten realizar las operaciones convencionales de edición de texto simple.
- Cuando en el editor hay un texto que ya ha sido analizado, todas las alteraciones que se intenten realizar sobre él serán filtradas por las funciones especiales definidas en ICEeditor.

El interés de esta sección se centra en el segundo caso, naturalmente, puesto que es el que está directamente relacionado con el mantenimineto de la consistencia entre la información de análisis y el texto del editor y en la que están involucradas las capacidades incrementales del analizador subyacente.

El usuario puede interactuar con todos los elementos que constituyen ICEeditor en cualquier orden. Para determinar cuándo se deben activar las funciones propias del editor se deben capturar los eventos de ratón.

El siguiente fragmento de código captura los eventos producidos por la pulsación del botón izquierdo del ratón en el editor:

```
(te-set-local-binding texteditor 'down0 'modify-parsered-text)
```

La función `modify-parsered-text` toma como argumento una instancia de un objeto {`textedit`} y realiza las siguientes acciones:

1. Terminar la operación de edición anterior.
2. Obtener la posición (en caracteres) en la que se pulsó el ratón sobre el texto.
3. Buscar el componente léxico cuyo texto está situado más cerca de dicha posición.
4. Se comprueba que dicho componente léxico no había sido borrrado con enterioridad, en cuyo caso se emite una advertencia sonora[24].
5. Si dicho componente léxico no había sido previamente borrado, entonces se establecen adecuadamente los valores de las variables de operación, dependiendo del tipo de opración que se vaya a realizar:
   (a) **Inserción**
      i. Se establece el texto inicial de la operación a la cadena nula y el nuevo texto al valor `()`[25].
      ii. Se sitúan los cursores `#:ICEeditor:te-begin-op-cursor` y `#:ICEeditor:te-end-op-cursor` y el cursor de inserción del editor al principio del texto correspondiente al componente léxico[26].
      iii. Se calcula la distancia horizontal y vertical existente entre el primer carácter y el último del componente léxico señalado por el usuario al pulsar con el ratón[27].
      iv. Se establece a cero la longitud del texto incluido inicialmente en la operación, mediante la asignación de este valor a la variable `#:ICEeditor:te-op-length`.
      v. Se cambia la forma del cursor del editor al valor contenido en `#:ICEeditor:cursor-for-insert`.
   (b) **Borrado**
      i. Se establece como texto inicial el texto almacenado en la TTLT y como texto final la cadena vacía.
      ii. La longitud de la operación se toma de la longitud del texto del componente léxico.
      iii. Se utiliza la selección `#:ICEeditor:te-delete-sel` para resaltar el texto borrado.

[24] Se ha optado por no permitir el *desborrado* de componentes léxicos. Para realizar una modificación sobre un componente léxico borrado el usuario debe inserta un nuevo componente léxico, justo antes o después del borrado, con el texto modificado.

[25] Para indicar que no hay un valor establecido. No se puede utilizar la cadena nula poruqe indicaría que se ha introducido un componente léxico sin texto.

[26] Esto es así porque se ha establecido la convención de que el componente léxico a insertar irá antes del componente léxico sobre el que se ha pulsado. Se podría haber tomado la convención de que la inserción se realizase después del componente léxico señalado, pero se ha considerado más conveniente la primera opción por resulatr más natural para el usuario.

[27] Esta información es necesaria para conocer, al finalizar la operación, la parte de la esrcutura de asociación de los componentes léxicos con el texto que deberemos reorganizar. Recordemos que los componentes léxicos puede incluir en su texto cualquier carácter, incluidos saltos de línea.

iv. La foma del cursor se cambia al valor almacenado en `#:ICEeditor:cursor-for-delete`.

(c) **Modificación**

i. Se establece el texto inicial de la operación como el texto almacenado en la TTLT. Esta es una acción correcta aunque se realicen múltiples modificaciones de un mismo componente léxico, puesto que en el parser incremental se parte del texto utilizado en el análisis anterior.

ii. Se establce el texto final al valor `()` para indicar que no se ha establecido aún ningún valor para esta variable. No se puede utilizar la cadena vacía por las mismas razones expuestas en el caso de la inserción.

iii. Se sitúa el cursor `#:ICEeditor:te-begin-op-cursor` en la posición inicial del texto correspondiente al componente léxico.

iv. Se sitúa el cursor `#:ICEeditor:te-end-op-cursor` en la posición siguiente al último carácter del texto del componente léxico.

v. Se almacena el desplazamiento horizontal y vertical existente entre el primer carácter y el último en las variables `#:ICEeditor:te-end-item-x` y `#:ICEeditor:te-end-item-y`, respectivamente.

vi. Se establece la longitud de la operación a la longitud del componente léxico.

vii. Se incluye en la selección `#:ICEeditor:te-modify-sel` el texto actual del componente léxico.

viii. Se cambia la forma del cursor a la almacenada en `#:ICEeditor:te-cursor-for-modify`.

(d) **información**

i. Se incluye dentro de la selección `#:ICEeditor:te-info-sel` el texto del componente léxico señalado por el usuario, para que aparezca resaltado en las fuentes y colores correspondientes a la operación de información.

ii. Se muestra al usuario una ventana con los datos relevantes del componente léxico.

iii. Se elimina el texto de la selección[28].

iv. Se cambia la forma del cursor a `#:ICEeditor:cursor-for-info`.

6. Se almacena en un array asignado a la variable `#:ICEeditor:current-op` la siguiente información que será utilizada por las otras funciones que tratan con la edición interactiva de componentes léxicos:

   - El número del componente léxico.
   - Las coordenadas horizontal y vertical de inicio del testo del componente léxico.
   - El texto inicial del componente léxico.
   - El texto final del componente léxico[29].

7. Se utiliza la forma `funcall` para realizar una llamada a la función *normal* utilizada en los editores `textedit` cuando tiene lugar un evento provocado por la pulsación del botón izquierdo del ratón.

[28] Sólo se da información de un componente léxico cada vez.

[29] Este valor sólo ha podido ser establecido, en este instante, para la operación de borrado, con valor de cadena nula

Con ello se consigue establecer toda la información necesaria para tratar correctamente el proceso de edición de los componentes léxicos. Dicho proceso consta de dos operaciones básicas:

- La inserción de caracteres en el texto de un componente léxico.
- El borrado de caracteres de un componente léxico, sean éstos de los originales del componente léxico o de los insertados posteriormente.

3.8.2 La inserción de caracteres en un componente léxico

Para tratar correctamente la inserción de nuevos caracteres en el texto de un componente léxico es preciso capturar los eventos producidos por la pulsación de alguna tecla en el teclado.

Cuando se produce uno de tales eventos se invoca a la función `select-inserted-text`. Esta función toma como argumento la instancia de {`textedit`} almacenada como componente `TEXTEDITOR` en la jerarquía de aplicaciones de ICEeditor y un símbolo que indentifica el tipo de modificación que se ha realizado sobre el editor de texto.

La función `select-inserted-text` realiza las siguientes acciones:

1. Comprobar que el carácter que se va insertar se encuentra en una posición válida, entendiéndose como válidas todas las posiciones comprendidas entre el primer carácter del texto actual del componente léxico y el último carácter del texto actual del componente léxico[30]. Este control es necesario puesto que `textedit` no proporciona ninguna manera de capturar los movimientos de las teclas del cursor. Tan sólo se pueden capturar los eventos del teclado cuando se modifica el texto del editor[31]. Para realizar esta comprobación es preciso realizar el siguiente conjunto de operaciones:
   (a) Si el parámetro `command` tiene como valor el símbolo `te-interactive-line-break`, entonces es que se está tratando de insertar un caracter de salto de línea. En tal caso se realizan las operaciones siguientes:
      i. Borrar el salto de línea.
      ii. Comprobar que la posición en la cual se está intentando insertar el carácter de salto de línea es una posición válida dentro del componente léxico. Esto se hace situando un cursor al comienzo del texto del componente léxico y desplazándolo a la derecha un número de posiciones igual al número de caracteres actuales del componente léxico.
      iii. Si es una posición válida, entonces se inserta realmente el salto de línea.
   (b) Si el parámetro `command` es diferente de `te-interactive-line-break`, es que se está inentando insertar un carácter ordinario[32]. En este caso se procede como sigue:

[30] Por texto actual se entiende aquel que resulta de aplicar inserciones y borrados válidos sobre el texto inicial del componente léxico y que se encuentra seleccionado utilizando los atributos gráficos correspondientes a la operación de edición que se está llevando a cabo.

[31] Se podría optar por capturar los eventos directamente de la ventana utilizando las funciones del Virtual Bitmap Display de LE-LISP, pero plantea problemas a la hora de establecer una adecuada colaboración con las tareas de edición del `textedit`.

[32] Es decir, un carácter que no implica salto de línea.

i. Se mueve el cursor de inserción una posición a la izquierda, para de este modo quedar situado en la posición en la que ralmente se insertó el carácter.
ii. Comprobar que dicha posición se encuentra dentro de los límites impuestos por el texto del componente léxico.
iii. Si es así, se retorna el cursor a su posición original.
iv. Si la posición no es válida, entonces se borra el carácter introducido, con lo cual el texto retorna al estado en que se encontraba antes de insertar el carácter. Se emite una señal audible para advertir al usuario de que la acción realizada no es válida.

2. Extender la selección para incluir el carácter insertado, si éste está en una posición válida, para que se integre consistentemennte con el aspecto gráfico de los demás caracteres involucrados en operaciones del mismo tipo.

3.8.3 El borrado de caracteres en un componente léxico

Para tratar correctamente el borrado de los caracteres de un componente léxico, es preciso capturar los eventos producidos como consecuencia de la pulsación de las teclas de borrado[33]. Estos eventos se capturan utilizando las siguientes llamadas a funciones:

```
(te-set-local-binding texteditor #\bs 'delete-character)
(te-set-local-binding texteditor #\del 'delete-character)
```

en donde `texteditor` se refiere a la instancia de {`textedit`} incluida en la jerarquía de componentes de ICEeditor.

La función `delete-character` toma como argumento una instancia de {`textedit`} y realiza las siguientes acciones:

- Comprueba que el carácter que se intenta borrar está en una posición válida, es decir, dentro de los límites del componente léxico que se está intentando editar. A diferencia de los que sucedía con la función `select-inserted-text`, que era llamada *después* de que se insertase un carácter, esta función es llamada *antes* de que se borre el carácter, por lo que el cursor está situado justo después del carácter que va a ser borrado.
- Si el carácter a borrar está en posición válida, se borra llamando a la función `char-delete-back` con el editor como único argumento. Si el carácter a eliminar está fuera de los límites del componente léxico, entonces se emite una señal sonora de advertencia y no se modifica el texto.

3.8.4 Terminación de una operación de edición de componentes léxicos

Uno de los momentos críticos en la actividad de ICEeditor ocurre cuando se termina de editar alguno de los componentes léxicos. Entonces hay que realizar las siguientes dos acciones:

- Actualizar la entrada correspondiente en el campo `operations` de la estructura `ICEeditor`.

[33] Que en un **`textedit`** son la teclas DEL y BACKSPACE. Ambas teclas funcionan de idéntico modo, borrando el carácter situado inmediatamente antes del cursor.

- Sincronizar el contenido del campo `TTLT` con el texto que se encuentra en el editor.

La primera de las operaciones es sencilla, pues simplemente hay que utilizar la información de operación, que se va actualizando conforme se modifica el texto del componente léxico involucrado en la operación.

La segunda operación es más compleja, puesto que implica una reorganización de la tabla de enlace entre los componentes léxicos y el texto. La forma en que se realiza esta actualización se describe adecuadamente el el capítulo 2, concretamente en las secciones 2.7.5 y 2.4.5.

De la realización de ambas tareas se encarga la función denominada `terminate-current-operation`.

Capítulo 4

Comunicación entre LE-LISP y C

En este capítulo se describen los mecanismos de comunicación de LE-LISP con procedimientos escritos en otros lengujes de programación. Las interfaces externas del LE-LISP no se limitan a un lenguaje concreto ni a un sistema operativo determinado, aunque esta discusión se centrará en el caso concreto de la interacción con el lenguaje C bajo entornos Unix, puesto que para realizar esta tesina ha sido necesario enlazar dinámicamente funciones C con programas escritos en LE-LISP.

4.1 Introducción

En ciertas ocasiones es preciso realizar llamadas desde LE-LISP a procedimientos externos escritos en otros lenguajes de programación, principalmente C o Fortran. Dichos procedimientos se convertirán en funciones LE-LISP normales a las que se les puede pasar argumentos de cualquier tipo y de las que se puede recuperar el resultado devuelto. Las principales causas de esta comunicación con otros lenguajes son:

1. La necesidad de comunicar programas escritos en LE-LISP con herramientas o productos comerciales que no disponen de una interfaz con LISP pero sí con otro lenguaje de programación que generalmente suele ser C en el caso de entornos Unix y Fortran o C en el caso de entornos VMS. Este caso se da, por ejemplo, cuando se necesita realizar consultas a una base de datos relacional o cuando se tiene que enlazar una analizador sintáctico escrito en LE-LISP con un analizador léxico generado mediante la herramienta Flex [Paxon 94].

2. Conseguir un mejor rendimiento en la realización de aquellas tareas para las cuales LE-LISP no está especialmente dotado, por ejemplo tareas de cálculo numérico que involucran una elevada cantidad de operaciones complejas en punto flotante. Para realizar esas tareas con mayor eficiencia puede ser recomendable acudir a procedimientos externos escritos en lenguajes como C, Fortran o incluso ensamblador. Lo mismo puede decirse en el caso de ciertas operaciones basadas en bits sobre pantallas de alta resolución.

En las siguientes secciones se describe cómo enlazar LE-LISP con programas en C bajo entornos Unix. Esta descripción es válida también para programas Fortran y ensamblador

que respeten las convenciones de las llamadas C bajo Unix[1]. Sin embargo, pueden surgir problemas en la comunicación con programas escritos en lenguajes como el Pascal estándar, debido a la declaración de los ficheros en la cláusula `program`. Una posible solución consiste en escribir un programa Pascal que llame al sistema LE-LISP como un procedimiento externo.

4.2 Enlaces estáticos

Para poder llamar a funciones C desde LE-LISP hay que enlazar las funciones C compiladas con el núcleo del sistema LE-LISP y asociar a funciones LE-LISP puntos de entrada a funciones C. Este enlace se puede crear bien estáticamente cuando se crea el sistema, o bien dinámicamente bajo el control del sistema.

Los enlaces estáticos se obtienen mediante la creación de un nuevo fichero binario ejecutable que contenga tanto el núcleo del sistema LE-LISP como los módulos en C. Este nuevo binario se puede usar para generar un nuevo sistema LE-LISP que puede ser instalado sobre cualquier máquina Unix. El principal incoveniente radica en la necesidad de volver a crear el sistema cada vez que se desee hacer un cambio en alguno de los módulos C, como ocurre en el caso de un generador de compiladores.

4.2.1 Creación de un nuevo sistema LE-LISP

Las operaciones que hay que realizar para construir un sistema LE-LISP estándar son las siguientes:

1. Copiar los ficheros del medio de distribución (diquetes o cinta magnética) al disco de la máquina. El directorio de instalación recomendado es `/usr/ilog`.

2. Hacer un `cd` al directorio del sistema[2] y ejecutar el comando `newdir` sin argumentos.

3. Ejecutar el comando `make` desde el directorio del sistema.

Para enlazar en modo estático funciones C es necesario realizar previamente un enlace permanente entre el fichero binario `lelispbin.o` y los módulos C correspondientes, generando un nuevo binario. El fichero `lelispbin.o` es el resultado de enlazar mediante `ld -r` todos los componentes del sistema LE-LISP excepto los módulos C, los cuales se obtienen compilando los ficheros C en el directorio `common`. Para facilitar el proceso conviene definir una nueva entrada en el fichero `makefile` para enlazar y configurar el nuevo sistema LE-LISP.

Por ejemplo, si se añaden las siguientes entradas en fichero `makefile` y se ejecuta el comando `make totolisp` desde el directorio del sistema, se obtendrá un nuevo sistema llamado `totolisp` que contendrá enlaces estáticos al múdulo `toto.o` y a la librería `totolib.a`. El fichero `totoconf.ll` será un pequeño programa LE-LISP con las instrucciones de configuración.

[1] Estas convenciones incluyen el particular modo en que el lenguaje C sitúa los argumentos y el valor devuelto en la pila.

[2] El directorio del sistema tiene la forma `/usr/ilog/lelisp/`*system*, donde *system* identifica la arquitectura de la máquina. Por ejemplo, en un IBM RS/6000 el directorio system será `/usr/ilog/lelisp/rs6000`.

```
totolisp: totolelispbin totoconf.ll
        config totolisp totolelispbin totoconf.ll

totolelispbin: toto.o
        cc -x -n $(cflags) lelispbin.o ../common/lelisp.c \
                           toto.o -ltotolib -lm -lc -o totolelispbin
```

4.2.2 Funciones de interfaz con C para enlaces estáticos

Las siguientes funciones se utilizan para llamar a funciones externas en lenguaje C ligadas mediante enlaces estáticos:

- `(getglobal` *proc*`)`. Esta función devuelve el punto de entrada[3] de un procedimiento externo llamado *proc*. Este nombre debe existir en la tabla de símbolos generada por el editor de enlaces cuando se creó el sistema LE-LISP. Hay que tener en cuanta que los nombre de símbolos están formados por el nombre de la función tal como se define en el lenguaje C precedida por un guión bajo[4].

- `(callextern` *direccion tipo* v_1 t_1 ... v_n t_n `)`. Esta función llama un procedimiento externo que tiene *dirección* como punto de arranque. El procedimiento retornará un valor de tipo *tipo*. Los v_i indican los argumentos de tipo t_i pasados al procedimiento. Los tipos t_i se codifican como sigue:

0	puntero
1	número entero
2	número de punto flotante
3	cadena de caracteres
4	vector de S-expresiones
7	vector de números enteros
8	vector de números de punto flotante

  Los códigos 5 y 6 no tienen importancia cuando se trata de funciones C, ya que se refieren al paso por referencia en Fortran de números enteros y números de punto flotante, respectivamente.

  El tipo *tipo* devuelto por la función externa llamada tan sólo puede ser de tipos 0, 1, 2 ó 3.

4.3 Enlaces dinámicos

Los enlaces estáticos tienen como inconveniente la necesidad de recompilar todo el sistema LE-LISP cada vez que se deba realizar algún cambio en una de las funciones C enlazadas estáticamente. Los enlaces dinámicos vienen a resolver este problema ya que permiten asociar dinámicamente funciones LE-LISP a procedimientos externos.

Sólo se pueden establecer enlaces dinámicos en aquella máquinas que posean editores de enlaces capaces de realizar un enlace incremental, es decir, capaces de ir estableciendo

[3] Dirección de comienzo.
[4] Underscore.

dinámicamente enlaces entre funciones a medida que son requeridos. Esta es la causa de que los enlaces dinámicos no estén disponobles en sistemas que corran bajo AIX o Solaris 2.x[5].

En aquellos sistemas en los que es posible realizar enlaces dinámicos, su utilización elimina la necesidad de volver a crear el sistema después de cambiar los módulos C. Además, los módulos enlazados dinámicamente se mantienen en los ficheros de imágenes de memoria, por lo que paradógicamente los enlaces dinámicos acaban siendo más perdurables que los estáticos.

4.3.1 Funciones de interfaz con C para enlaces dinámicos

Las siguientes funciones están relacionadas con las llamadas a funciones externas escritas en leguaje C y enlazadas dinámicamente con el sistema LE-LISP:

- (`defextern` *símbolo ltipo tipo*). Esta función asocia dinámicamente una función LE-LISP con un procedimiento externo, de tal modo que el procedimiento externo de nombre *símbolo* se convertirá en una nueva función LE-LISP. Los tipos de los argumentos a dicha función externa se indican en la lista *ltipo*, mientras que *tipo* indica el tipo del valor devuelto.
- (`cload` *string*). El argumento *string* es el nombre del módulo compilado que va a ser enlazado dinámicamente. Esta función llama al editor de enlaces `ld` de Unix para enlazar el módulo C con el sistema LE-LISP, de tal modo que el código compilado de dicho módulo se carga en la zona de memoria de código del LE-LISP, lo que a su vez permite que sea salvado en la imagen de memoria del LE-LISP.

Para ilustrar los pasos necesarios para realizar enlaces dinámicos vamos a suponer que se tiene un módulo llamado `toto.c` escrito en lenguaje C. Desde el propio entorno LE-LISP podemos compilar `toto.c` haciando una llamada al shell mediante la utilización del carácter macro `!`[6]:

```
? !cc -c toto.c
= t
```

El siguiente paso consiste en enlazar el módulo `toto.o` al sistema:

```
? (cload "toto.o")
= (18 . 2684)
```

La función `cload` pasa su argumento, sin realizar ningún cambio en él, al editor de enlaces `ld`. Esto permite emplear cualquier opción válida para `ld`, incluyendo las que permiten cargar librerías o enlazar a la vez varios módulos. A continuación se muestran dos ejemplos:

- Para cargar el punto de entrada correspondiente a la función de la librería matemática que calcula el seno hiperbólico se emplearía la siguiente llamada a `cload`:

[5] En el sistema operativo Solaris 2.x (SunOS 5.x) se ha eliminado la opción de enlace incremental `-A` del editor de enlaces `ln`. Dicha opción sí se encuentra disponible bajo Solaris 1.x (SunOS 4.1.x). Tengo noticias de que el investigador holandés Casper Dik ha realizado modificaciones en el código fuente de LE-LISP para transformar las llamadas al editor de enlaces en llamadas a las nuevas funciones `dlopen` y `dlclose` de Solaris 2.x. Se puede contactar con él mediante e-mail en `casper@fwi.uva.nl`.

[6] El carácter macro ! ejecuta el comando que se le pasa como argumento, utilizando el shell `/bin/sh`.

```
(cload "-u _sinh -lm")
```

- Para cargar juntos los módulos `toto.o`, `pepe.o` y `foo.o` se utilizaría la siguiente llamada:

```
(cload "toto.o pepe.o foo.o")
```

En general, la llamada(`cload` *string*) generará la siguiente llamada al aditor de enlaces[7]:

`ld -A` *lelispbin* `-Bstatic -N -x -T` *fin* `-o` *temporal string* `-lc`

donde *lelispbin* se corresponde con el path que contiene el directorio donde está localizado el binario del sistema LE-LISP que se está ejecutando, *fin* es la primera dirección disponible en la zona de LE-LISP reservada para código compilado y *temporal* es un fichero temporal único en el directorio `/tmp`.

Una vez que se ha realizado el enlace, el fichero *temporal* contendrá el código del módulo C, que se cargará en la zona `code` de LE-LISP. Este fichero temporal se utilizará en lugar de *lelispbin* en las siguientes llamadas a `cload`. De este modo, se pueden realizar numerosas llamadas a `cload` para modificar secuencialmente una imagen ejecutable.

4.3.2 Correspondencias de tipos en los enlaces dinámicos

En una llamada (`defextern` *símbolo ltipo tipo*), la lista *ltipo* puede contener los siguientes tipos:

`t`	un puntero LE-LISP
`external`	un puntero externo
`fix`	un número entero
`float`	un número de punto flotante
`string`	una cadena de caracteres
`vector`	un vector de S-expresiones
`fixvector`	un vector de enteros
`floatvector`	un vector de floats

El tipo *tipo* devuelto debe ser alguno de los siguientes:

`t`	un puntero LE-LISP
`external`	un puntero externo
`fix`	un número entero
`float`	un número de punto flotante
`string`	una cadena de caracteres

A continuación se examina cada uno de estos tipos más detalladamente:

- El tipo `t` se corresponde con cualquier puntero LE-LISP que es pasado sin alteraciones. De la experiencia práctica se deduce que es el modo más seguro de pasar punteros hacia y desde funciones C.

[7] En una máquina Sun4 bajo Solaris 1.1 (SunOS 4.1.3).

- El tipo `external` corresponde a los punteros externos al espacio de memoria de LE-LISP. Está representado por una dirección LE-LISP. Generalmente, un puntero de este tipo es el resultado de una llamada a un procedimiento externo. El resultado de dicha llamada suele ser almacenado y pasado a otro procedimiento externo.
- El tipo `fix` se corresponde con un valor de tipo numérico entero.
- El tipo `float` se corresponde con un valor numérico de punto flotante. Este tipo trabaja bien tanto en el modo LE-LISP de punto flotante de 32 bits como en el de 64 bits.
- Un valor de tipo `string` se refiere a la dirección del primer carácter de una cadena de caracteres. No se puede pasar una cadena de caracteres por valor.
- Un valor de tipo `fixvector` indica la posición del primer elemento de un vector de números enteros. Los vectores de enteros siempre se pasan por referencia. Debido al modo en que LE-LISP representa los enteros, un vector de enteros es diferente de un vector de S-expresiones que contenga sólo enteros.
- El tipo `floatvector` es como el `fixvector` sólo que referido a vectores numéricos de punto flotante en 32 bits. No permite la utilización de números de punto flotante de 64 bits. Un vector de floats es diferente de un vector de S-expresiones conteniendo sólo floats.

En la tabla 4.1 se muestran las conversiones que tienen lugar cuando se pasan argumentos a funciones C (columna 'Argumento') y cuando éstas devuelven sus resultados (columna 'Resultado'). La columna etiquetada como 'Tipo LE-LISP' muestra el tipo definido en `defextern` mientras que la columna etiquetada 'Tipo C' muestra el tipo de los argumentos recibidos por la función C conectada a LE-LISP[8].

Los punteros LE-LISP son, o bien punteros a la zona de datos de LE-LISP, o bien enteros cortos de 16 bits. El fichero `lelisp.h` contiene las declaraciones del tipo de estructura C a las cuales apuntan los punteros LE-LISP. Respecto al tipo `external` decir que no es cierto que pueda manipular sin más cualquier puntero del sistema LE-LISP, sino que primero debe ser convertido en un `cons` de dos enteros que representen las mitades más significativas y menos significativas de la dirección.

4.3.3 El fichero `lelisp.h`

El fichero `lelisp.h` se encuentra dentro de la distribución estándar de LE-LISP y contiene las estructuras C necesarias para trabajar con objetos LE-LISP desde funciones escritas en lenguaje C. Se debe incluir la directiva `#include "lelisp.h"` en todos aquellos programas que precisen manipular objetos LE-LISP, aunque dicha manipulación sea una actividad desaconsejada por los propios constructores del sistema LE-LISP.

A continuación se muestra una lista de las declaraciones de estructuras y tipos más interesantes incluidas en `lelisp.h`:

- El tipo `LL_OBJECT` es un tipo polimórfico que cubre todos los tipos de objetos LE-LISP, como por ejemplo `LL_SYMBOL` y `LL_CONS`:

[8] `any *` significa 'puntero a cualquier tipo C'.

Tabla 4.1: Conversiones de tipos entre LE-LISP y C.

Tipo LE-LISP	Tipo C	Argumento.	Resultado
`fix`	`int`	Signo extendido a 32 bits.	Covertido a 16 bits.
`float`	`double`	Se pasa el valor.	Creado float LISP.
`string`	`char *`	Se pasa un puntero a `char`.	Asignado string LISP. String C copiado en él.
`vector`	`any * []`	Se pasa un array de punteros LISP.	LE-LISP recibe el puntero C sin cambios.
`fixvector`	`int * []`	Se pasa un array de enteros. El signo de cada uno se extiende a 32 bits.	No implementado, pero los cambios hechos al array C son recuperados por LE-LISP.
`floatvector`	`float * []`	Se pasa un array de floats.	No implementado, pero los cambios hechos al array C son recuperados por LE-LISP
`t`	`any *`	C recibe el puntero sin cambios.	LE-LISP recive el puntero sin cambios.
`external`	`any *`	El objeto LE-LISP debe ser un `cons` de dos enteros codificando una dorección.	Se asigna un `cons` LE-LISP que se inicializa con las partes más y menos significativas de la dirección del objeto C. La función devuelve el `cons` como su resultado.

```
typedef char *LL_OBJECT;
```

- La estructura `LL_CONS` permite manipular objetos `cons` desde C:

```
struct LL_CONS {
  LL_OBJECT ll_car;
  LL_OBJECT ll_cdr;
};
```

- La estructura `LL_SYMBOL` permite manipular símbolos LE-LISP desde funciones C. El significado de los distintos campos se explica en la sección B.1.1, página 169. Los campos `ll_alink` y `ll_pname` no se deben tocar en ningún caso.

```
struct LL_SYMBOL {
  LL_OBJECT ll_cval;
  LL_OBJECT ll_plist;
  LL_OBJECT ll_fval;
  LL_OBJECT ll_alink;
  LL_OBJECT ll_pkgc;
  LL_OBJECT ll_oval;
  char      ll_ftype;
  char      ll_ptype;
  short     ll_pad;
  LL_OBJECT ll_pname;
};
```

- Las cadenas de caracteres y los vectores son punteros a punteros sobre estructuras de tamaño variable. La doble indirección resulta necesaria para gestionar la desasignación de memoria en el proceso de recolección de basura[9]. Puesto que en C no existen estructuras de tamaño variable, las declaraciones de `LL_STRING` y de `LL_VECTOR` corresponden a objetos de 1 byte[10]. Los campos `ll_strsize` y `ll_vecsiz` indican el tamaño real de los objetos[11]. No se deben modificar los campos `ll_strobj`, `ll_strarr`, `ll_strtyp`, `ll_vecobj`, `ll_vecarr` ni `ll_vectyp`.

```
struct LL_STRING {
  struct {
    struct LL_STRING *ll_strarr; /* enlace a otros strings */
    int              ll_strsiz;  /* numero de caracteres */
    char             ll_strfil;  /* direccion del primer
                                  * caracter */
    } *ll_strobj;
  LL_OBJECT ll_strtyp;           /* puntero al simbolo LE-LISP */
};


struct LL_VECTOR {
  struct {
    struct LL_VECTOR *ll_vecarr; /* enlace a otros vectores */
    int              ll_vecsiz;  /* numero de elementos */
    LL_OBJECT        ll_vecfil;  /* direccion del primer
                                  * elemento */
  } *ll_vecobj;
  LL_OBJECT ll_vectyp;           /* puntero al simbolo LE-LISP */
};
```

- La macro C llamada `LL_C_FLOAT` transforma un número de punto flotante en LE-LISP en el correspondiente número de punto flotante en C.
- Las funciones relacionadas con las llamadas al sistema LE-LISP desde funciones C se muestran en la sección 4.4, página 73.

4.3.4 Un pequeño ejemplo

Para ilustrar el enlace dinámico de funciones C con LE-LISP vamos a mostrar un pequeño ejemplo, consistente en la creación de funciones en C que permitan acceder a un programa escrito en LE-LISP a los campos de una estructura compleja creada en C[12].

El fichero `def.h` contiene la definición de las estructuras `object_values` y `VALUE`:

```
#include <stdio.h>

struct VALUE
{
```

[9] Garbage collection.
[10] Strings con un sólo carácter o vectores con un único elemento.
[11] Número de caracteres en un string y número de elementos en un vector.
[12] El compilador de C utilizado es el `cc` estándar incluido en la distribución de SunOS 4.1.3. La sintaxis utilizada se corresponde con el lenguaje C clásico K&R [Kerninghan y Ritchie 85].

```
  int data_item;
  VALUE *next_value;
};

struct object_values
{
  VALUE * level,
          temperature,
          pressure;
},
```

En el fichero `example.c` se define una variable de tipo `object_values` llamada `myvalues` que almacena tres listas con los datos de nivel, temperatura y presión en varios instantes de tiempo. También se definen las funciones `car_level`, `car_temperature` y `car_pressure` que permiten obtener el primer valor de la lista de niveles, temperaturas y pressiones asociadas a un objeto de tipo `object_values`. La función `initialize_values` da valores a los campos de una variable de tipo `object_values`. Por simplicidad sólo crea el primer elemento de cada lista de valores.

```
#include <stdio.h>
#include <malloc.h>
#include "def.h"

struct object_values myvalues={NULL,NULL,NULL};

/* return the address of 'myvalues' */
struct object_value *
get_object()
{
  return &myvalues;
}

/* Initialize the values of the object returned by 'get_object()' */
struct object_values *
initialize_values(a_object_value)
  struct object_values * a_value_object;
{
  struct VALUE *level;
  struct VALUE *temperature;
  struct VALUE *pressure;

  /* initialize level */
  level=malloc(sizeof(struct VALUE));
  level->data_item=46;
  level->next_value=NULL;
  (a_object_value->level)=level;

  /* initialize temperature */
  temperature=malloc(sizeof(struct VALUE));
  temperature->data_item=28;
  temperature->next_value=NULL;
  (a_object_value->temperature)=temperature;

  /* initialize pressure */
```

```
  pressure=malloc(sizeof(struct VALUE));
  pressure->data_item=770;
  pressure->next_value=NULL;
  (a_object_value->pressure)=pressure;

  /* return the initialized object */
  return a_value_object;
}

/* return the first level of 'a_object_value' */
integer
car_level(a_object_value)
  struct object_values a_object_value;
{
  return (a_object_value->level)->data_item;
}

/* return the first temperature of 'a_object_value' */
integer
car_temperature(a_object_value)
  struct object_values a_object_value;
{
  return (a_object_value->temperature)->data_item;
}

/* return the first pressure of 'a_object_value' */
integer
car_pressure(a_object_value)
  struct object_values a_object_value;
{
  return (a_object_value->pressure)->data_item;
}
```

Mediante `cc -c example.c` obtenemos el fichero binario `example.o` con las funciones C compiladas.

En el fichero `example.ll` se crean mediante `defextern` los enlaces dinámicos a las funciones contenidas en `example.o`:

```
(cload "example.o")

(defextern _get_object        ()  t)
(defextern _initialize_values (t) t)
(defextern _car_level         (t) fix)
(defextern _car_temperature   (t) fix)
(defextern _car_pressure      (t) fix)

(setq myvalues (_initialize_values (_get_object)))

(print "The level is      : " (_car_level       myvalues " %")
(print "The temperature is: " (_car_temperature myvalues " celsius grades")
(print "The pressure is   : " (_car_pressure    myvalues " mmHg")
```

Una vez dentro del sistema LE-LISP podemos ejcutar el programa `example.ll` para obtener la siguiente respuesta:

```
? ^Lexample.ll
The level is       : 46 %
The temperature is: 28 celsius grades
The pressure is    : 770 mmHg
```

4.4 Llamando al sistema LE-LISP desde C

En ciertas máquinas, entre las que se encuentran las Sun4, se puede llamar al sistema LE-LISP desde una función C que a su vez había sido llamada desde LE-LISP. Este proceso consta de los tres pasos siguientes:

1. Se llama al sistema LE-LISP.
2. Desde LE-LISP se llama a una función C enlazada dinámicamente.
3. Dicha función C llama al sistema LE-LISP.

El límite de la profundidad de estas llamadas mutuas lo marca el agotamiento de los recursos del sistema.

Las funciones C llamadas desde LE-LISP se deben declarar mediante `defextern`. Desde dichas funciones C se pueden utilizar las siguientes funciones definidas en el fichero `lelisp.h`:

- `getsym`, que toma como argumento un string y devuelve el símbolo LE-LISP que tiene como `p-name` dicho string:

```
struct LL_SYMBOL *
getsym(s)
  char *s;
{
... /* codigo de getsym */
};
```

- `pusharg`, que dados un tipo y un valor de dicho tipo, situa dicho valor en la cima de la pila (stack):

```
pusharg(type, val)
  int type;
  any val;
{
... /* codigo de pusharg */
}
```

- `lispcall` permite llamar a una función LE-LISP. Toma como argumentos el tipo del valor devuelto por la función LE-LISP, el número de argumentos puestos en la pila por medio de `pusharg` y la propia función LE-LISP que se desea llamar:

```
LL_OBJECT lispcall(typeres, narg, function)
  int typeres;
  int narg;
  struct LL_SYMBOL *function;
{
... /* codigo de lispcall */
}
```

Los tipos de los argumentos para `pusharg` así como el valor devuelto por `lispcall` deberá ser `LLT_FIX`, `LLT_FLOAT`, `LLT_STRING`, `LLT_VECTOR` o `LL_T`[13].

Los pasos generales a seguir para llamar a una función LE-LISP desde una función escrita en lenguaje C son los siguientes:

1. Obtener el símbolo correspondiente a la función LE-LISP mediante una llamada a `getsym`.
2. Situar en la pila los argumentos de la función LE-LISP utilizando para ello la función `pusharg`. El orden en que se realizan estas operaciones coincide con la posición de los argumentos de la función LE-LISP que se va a llamar, es decir, primero se sitúa el primer argumento de la función LE-LISP, después el segundo, etc.
3. Llamar a la función LE-LISP mediante `lispcall`.

Se permite la realización de llamadas recursivas mutuas entre funciones C y funciones LE-LISP, lo que muestra el alto grado de integración que se puede conseguir entre ambos lenguajes.

En ciertas máquina, incluso se puede pasar el control de una aplicación LE-LISP a una función C. En la distribución de LE-LISP para algunas versiones de Unix se proporciona un fichero `llmain.c` cuya misión es la de actuar como programa principal para lanzar el sistema LE-LISP. El usuario puede ejecutar todo el código C que desee antes de lanzar LE-LISP. Posteriormente, se puede llamar a funciones C utilizando `defextern` y el proceso de enlace dinámico o bien retornar al programa principal en C. Este mecanismo se implementa mediante las llamadas al sistema de Unix `setjmp` y `longjmp`.

[13] El tipo `LL_T` representa un puntero a cualquier cosa.

Capítulo 5

El análisis léxico

En este capítulo se tratan los aspectos correspondientes al analizador léxico. La discusión se centrará en los dos problemas siguientes: la consecución de un generador de analizadores léxicos que exhiban un comportamiento no determinista en el reconocimiento de los componentes léxicos y la integración de dichos analizadores con ICEeditor.

5.1 Descripción del problema

En la tarea del tratamientos de lenguajes, se reserva al analizador léxico un papel muy importante, pues será el encargado de reconocer las *palabras* que posteriormente serán pasadas a los analizadores sintáctico y semántico.

En el análisis de los lenguajes de programación, que es el entorno más común en el cual se hace uso de los analizadores léxicos, cada palabra[1] del lenguaje se corresponde únivocamente con un sólo componente léxico posible. Por ejemplo, en un lenguaje como el Pascal la palabra clave `IF`[2] se hace corresponder con un único componente léxico[3]. Entre cada palabra clave del lenguaje y cada componente léxico existe un relación biyectiva[4]. Lo mismo puede decirse del resto de los lenguajes de programación.

Sin embargo, en el contexto del procesamiento de lenguajes naturales, una misma palabra puede corresponderse con varios componentes léxicos distintos. Esto quiere decir que para un componente léxico dado, su materialización en el texto, la cadena de caracteres que se debe escribir para que sea reconocido como tal componente léxico, no es exclusiva, sino que pueden existir otros componentes léxicos cuyo texto coincide con el dado. Un ejemplo en castellano lo tenemos en la palabra `casa`, que puede ser reconocida como dos formas verbales distintas[5] o como un sustantivo.

Para solucionar este problema, el analizador léxico debe mostrar un comportamiento no determinista, esto es, debe dar todos los posibles reconocimientos para una palabra dada y no debe pasar a reconocer un nuevo componente léxico hasta que no se han agotado

[1] Consideramos una palabra como el conjunto de caracteres delimitado por separadores, es decir, el conjunto de caracteres cuya presencia en el texto fuente va a dar lugar al reconocimiento de un componente léxico por parte del analizador léxico.

[2] La cadena de caracteres "IF" en el código fuente del programa, siempre que no esté dentro de un string.

[3] Generalmente, cada componente léxico se suele identificar mediante un número entero, que es el valor que se pasa al analizador sintáctico.

[4] Los demás identificadores que aparecen en el código fuente (nombres de variables, de funciones, de tipos,...) se resuelven mediante la utilización de tablas de símbolos [Aho et al. 90].

[5] La tercera persona del presente de indicativo y la segunda persona del imperativo.

todas las posibilidades para el componente léxico precedente. Sólo de esta manera se puede asegurar que el posterior análisis sintáctico y semántico será capaz de analizar correctamento el texto, seleccionando de entre todas las opciones a nivel léxico, aquellas que concuerdan con el resto de la estructura sintáctica del texto.

Otra diferencia con respecto a los lenguajes de programación está en la forma en que se reconocen los componentes léxicos. En éstos últimos cada palabra clave o identificador tiene una única forma escrita, sin variaciones, y su número es limitado[6]. Sin embargo, en una lengua como el castellano existen decenas de miles de *lemas*, que se corresponden con las entradas en el diccionario. Cada lema se corresponde con lo que podríamos llamar la forma canónica de un conjunto de palabras. Por ejemplo, el lema de un sustantivo lo constituye la forma masculina singular. Las demás formas (el femenino y los plurales en masculino y singular) pueden ser derivadas a partir de la forma canónica utilizando las reglas léxicas correspondientes al tipo de sustantivo de que se trate[7]. En el caso de los verbos la utilización de reglas léxicas es mucho más acusada. En los verbos regulares, nos basta con conocer la forma del infinitivo para poder conjugar todos los posibles tiempos verbales. Los verbos irregulares no son totalmente irregulares, sino que pueden ser agrupados en una serie de grupos verbales (en torno a la treintena) que siguen sus propias reglas léxicas, aunque puede ser preciso utilizar más de una raiz para su conjugación[8].

Una forma de solucionar los problemas planteados consiste en utilizar una inmensa base de datos en la cual estén almacenadas todas las formas de todas las palabras. Cada vez que se tenga que reconocer una palabra, se accede a esta base de datos y se recuperan todas las posibles alternativas. Este enfoque presenta serios inconvenientes:

- El tamaño desproporcionado que debería tener la base de datos para ser capaz de obtener buenos resultados con textos grandes, obliga a que resida en almacenamiento secundario. La necesidad de acceder continuamente al disco degrada mucho el rendimiento del sistema.

- Cada vez que se encuentre una nueva palabra, se deben introducir todas las formas derivadas de su lema, si queremos que la base de datos sea consistente. Por ejemplo, para cada verbo es necesario introducir entera toda su conjugación. Se puede evitar la entrada manual mediante un generador de formas derivadas, pero aún así se tiene el problema del crecimiento exponencial del tamaño de la base de datos con respecto al número de lemas, lo que degrada el rendimiento ya que introduce retardos en el tiempo de acceso, siempre considerando que se posee una buena estructura de índices.

- No hace uso del conocimiento, expresado en forma de reglas léxicas, que se posee del léxico.

Para evitar esos inconvenientes hemos considerado que la mejor solución es utilizar un generador de analizadores léxicos en el que se puedan incorporar las reglas léxicas y el comportamiento no determinista. El resultado es un programa ejecutable que se encarga de realizar las labores de análisis léxico en un tiempo muy pequeño y con un bajo consumo de memoria.

[6] A lo sumo de unos pocos cientos.

[7] Los sustantivos más comunes hacen el femenino añadiendo al lexema la terminación **a**. Para construir los plurales se utilizan las terminaciones **os** y **as**.

[8] Un caso típico es el del verbo **pensar**, para cuya conjugación es necesario utilizar las raices **pens** y **piens**.

Se ha optado por utilizar el generador de analizadores léxicos llamado Flex[9] [Paxon 94]. Esta herramienta es un derivado de Lex, el generador de analizadores léxicos estándar incluido en el sistema operativo Unix. Flex presenta como ventajas sobre Lex una mayor velocidad[10] y una mayor flexibilidad, conservando la compatibilidad. Cualquier programa escrito en Lex funcionará sin problemas con Flex. Sin embargo, un programa escrito para Flex que haga uso de las características ampliadas de éste, no podrá ser compilado con Lex, como resulta evidente.

5.2 Las reglas léxicas

Las palabras de un idioma como el castellano se pueden dividir en dos partes: el lexema y los sufijos. El lexema es la parte invariante de la palabra. En ciertos casos, la palabra está formada únicamente por el lexema. Tal es el caso de las preposiciones. Sin embargo, en la mayoría de los casos las palabras se forman mediante la aposición de una serie de sufijos al lexema, como ocurre, por ejemplo, en el caso de los sustantivos, los adjetivos y los verbos. Cuando de un lexema se puede obtener un conjunto de palabras que derivan de él, se toma una de esas palabras para representar el conjunto: es lo que se denomina el *lema* de ese conjunto de palabras. En el caso de los sustantivos y los adjetivos el lema lo constituye la forma masculina singular, y en el de los verbos el infinitivo.

En ciertos casos, como en el de los verbos irregulares, las cosas no son tan simples, puesto que no todas las formas de la conjugación tienen la misma raiz. Pensemos por ejemplo en el verbo `pensar`. Parte de la conjugación utiliza como raiz `pens` (por ejemplo, las formas `pensé`, `pensaré`, etc.), pero otras utilizan `piens` como raiz (`pienso`, `piensas`, etc.). Sin embargo, todas las formas de la conjugación tienen un único lema: `pensar`.

Por lo tanto, a la hora de reconocer las palabras válidas del castellano se deberá partir del reconocimiento de los lexemas para posteriormente ir aplicando las reglas léxicas que indican cómo se formarán las palabras pertenecientes al conjunto de un determinado lema.

5.2.1 Ejemplos de reglas léxicas

Para aclarar el significado de las reglas léxicas utilizaremos como ejemplo los sustantivos. En castellano hemos detectado hasta 17 formas distintas de formar el género de una palabra, catalogadas como $G1$, $G2$,...,$G17$. Hay grupos muy comunes, como por ejemplo el $G1$, que añade una `o` en el caso del masculino y una `a` en el del femenino. Otros son mas inusuales, como el grupo $G7$, que forma el masculino añadiendo la terminación `que` y el femenino mediante la terminación `ca`. Un ejemplo de este grupo lo constituyen las palabras `cacique` y `cacica`.

Para formar el número se han detectado 9 formas, catalogadas como $N1$, $N2$,...,$N9$. El grupo más común es el $N1$, que añade una `s` para formar el plural y ningún carácter para el singular. Un grupo bastante inusual es el $N4$, que utiliza la terminación `c` para el singular y la `ques` para el plural. Es el caso de las palabras `frac` y `fraques`.

La mayoría de los sustantivos forman tanto el género como el número, por lo que es necesario combinar los dos tipos de reglas léxicas. En cualquier caso, siempre se forma

[9] La versión utilizada ha sido la 2.4.6 de Enero de 1994.
[10] El nombre de Flex proviene de *Fast Lex*.

primero el género y después el número. No son válidas todas las combinaciones de grupos de formación género-número. Por ejemplo, los lexemas que utilizan el grupo $G1$ para formar el género siempre utilizan el grupo $N1$ para el número.

En el caso de los verbos, existen muchas más reglas léxicas que determinan cómo se construye cada una de las conjugaciones y la treintena de grupos de verbos irregulares.

Todo este conocimiento debe ser incorporado en la etapa de análisis léxico para determinar los componentes léxicos que se han reconocido. Puesto que las reglas léxicas son fijas, para ampliar posteriormente el conjunto de palabras que es capaz de reconocer el analizador tan sólo es necesario introducir el lexema y establecer los grupos con los cuales se forman las palabras válidas que comparten dicho lexema. Estas modificaciones se ven también facilitadas por el hecho de utilizar un generador de analizadores, en lugar de programar directamente en LISP o en cualquier otro lenguaje de programación de propósito general. En la gramática de Flex, añadir nuevas palabras consiste simplemente en añadir nuevos patrones en la parte izquierda de las reglas.

5.2.2 Las condiciones de arranque

Para facilitar la implementación de las reglas léxicas, se pueden utilizar las *condiciones de arranque*[11] de Flex, que permiten continuar la búsqueda de expresiones regulares una vez que ha sido detectado el lexema de una palabra.

Las condiciones de arranque proporcionan un mecanismo para establecer la ejecución condicional de las reglas. Cualquier regla cuyo patrón[12] comience por $<sc>$, se activará solamente cuando la condición de arranque *sc* esté activa.

Existen dos tipos de condiciones de arranque:

- *Inclusivas*, que se declaran mediante `%s` en la sección de declaraciones del programa Flex. Se caracterizan porque cuando se activan, también están activas todas las reglas que no comienzan por una condición de arranque.

- *Exclusivas*, que se declaran mediante `%x`. Cuando se activa una de estas condiciones de arranque, sólo permanecen activas aquellas reglas que comiencen con la misma condición.

Una condición de arranque *sc* se activa mediante la acción `BEGIN(`*sc*`)`.

5.2.3 Implementación de las reglas léxicas

El procedimiento mediante el cual se pueden implementar en Flex las reglas léxicas es el siguiente:

1. Utilizar reglas sin condición de arranque para reconocer los lexemas, los cuales constituirán los patrones de tales reglas.

2. Definir una condición de arranque para cada grupo de reglas léxicas. Utilizar los sufijos como patrones de las reglas incluidas en esa condición.

[11] *Start conditions.*
[12] La parte izquierda de la regla.

3. En las acciones que se disparan por el reconocimiento de un lexema, incluir un `BEGIN` a la condición de arranque que represente al grupo en el cual se crean las palabras derivadas de ese lexema.
4. En las acciones de las reglas con condiciones de arranque se puede tomar una de las siguientes alternativas:
   (a) Incluir un `BEGIN` a una condición de arranque que indique los sufijos que deben ser reconocidos a continuación. Esto se utiliza, por ejemplo, cuando después de reconocer el género de un sustantivo es preciso reconocer su número.
   (b) Si no es necesario comprobar la existencia de más sufijos, dar por terminado el reconocimiento de la palabra.

Todas las condiciones de arranque deberán ser exclusivas, puesto que que con ello evitamos el problema de un posible reconocimiento de una palabra dentro de otra.

Es conveniente considerar la existencia de una condición `ERROR`, a la que se envíe el analizador cuando una palabra no pueda ser reconocida[13]. Cuando el analizador entra en este estado de error, debe actuar en *modo pánico*[14] [Aho et al. 90], saltando todos los caracteres siguientes hasta que encuentra un carácter de sincronización, que en nuestro caso será un separador.

La utilización de condiciones de arranque es similar a definir un *miniautómata* para examinar las reglas léxicas.

5.2.4 Un ejemplo sencillo

Para que se pueda comprender mejor la implementación de las reglas léxicas, vamos a mostrar un pequeño ejemplo. Se trata de reconocer los sustantivos que forman el género mediante $G1$ y el número mediante $N1$.

Lo primero que debemos hacer es definir las condiciones de arranque en la sección inicial de declaraciones del programa Flex. Para nuestro ejemplo, es suficiente con definir $G1$, $N1$ y E. esta última constituye la condición de error a la cual se enviará el autómata del reconocedor siempre que una palabra no puede ser reconocida. Esto se traduce en código de la siguiente manera[15]:

```
/* Error */
%x E

/* gender */
%x G1

/* Number */
%x N1
```

Después definimos los dos patrones siguientes:

- `sep`, formado por los caracteres separadores. Estos caracteres se utilizan para evitar reconocer una palabra dentro de otra. El conjunto de caracteres separadores está formado por los siguinetes elementos:

[13] Puede ser que no se reconozca el lexema como válido o que se reconozca el lexema pero no se pueda aplicar ninguna de las reglas léxicas para construir la palabra.

[14] *Panic mode* en la literatura anglosajona.

[15] Los comentarios en Flex se ponen a la manera del C, esto es, acotados entre un `/*` y un `*/`.

  - Espacio.
  - Tabulador.
  - Coma.
  - Punto y coma.
  - Dos puntos.
  - Punto.
  - El carácter de nueva línea.

- `nosep`, en el que se engloban todos los caracteres que no son separadores y que por tanto pueden aparecer dentro de una palabra.

En las siguientes línea de código se definen estos dos patrones[16]:

```
/* Separator characters */
sep   [" "\t,;:\.\n]

/* Non-separator characters */
nosep [^" "\t,;:\.\n]]
```

Con esto ya podemos comenzar a definir reglas en la sección de reglas del programa Flex. Como ejemplo vamos a mostrar cómo se reconocen las palabras cuyos lemas son `perro` y `gato`.

Los lexemas de estas palabra son `perr` y `gat`, respectivamente. Como dijimos anteriormente, deberán ser utilizados como patrones en la parte izquierda de una regla sin condición de arranque, como se muestra a continuación:

```
gat    |
perr   {yymore();
        BEGIN G1;
       }
```

Mediante la llamada a la función `yymore` indicamos a Flex que queremos seguir añadiendo en `yytext`[17] los caracteres que sean reconocidos a continuación.

Ahora es preciso definir la reglas mediante las cuales se reconoce el género de las palabras incluidas en el grupo $G1$. Estas palabras forman el masculino añadiendo una `o` al lexema y el femenino añadiendo una `a`. En el siguiente código se muestra la definición de tales reglas:

```
/* Terminations for gender: Case 1 */
<G1>o          {/* Masculine */
                printf(" masculine");
                yymore();
                BEGIN N1:
               }
<G1>a          {/* Feminine */
                printf(" feminine");
                yymore();
                BEGIN N1;
               }
```

[16] El carácter \ se utiliza en Flex para *escapar* aquellos caracteres que tienen un significado especial en la gramática flex. Esto ocurre por ejemplo con el punto.

[17] En la variable **yytext** se almacena el texto reconocido en cada regla.

El prefijo <G1> antes de los patrones `o` y `a` indica que ambas reglas sólo se activarán cuando la condición de arranque `G1` haya sido activada mediante `BEGIN G1`. Como `G1` ha sido definida en modo exclusivo, después de `BEGIN G1` únicamente estas dos reglas estarán activas.

Como todas las palabras que forman el género en $G1$ forman el número en $N1$, ambas reglas utilizan `BEGIN N1` para activar la condición de arranque `N1`.

Las reglas para el reconocimiento del número son la siguientes:

```
/* Terminations for number: Case 1 */
<N1>s/{sep}    {/* Plural */
                printf(", plural --> %s",yytext);
                BEGIN(INITIAL)
               }
<N1>""/{sep}   {/* Singular */
                printf(", singular --> %s", yytext);
                BEGIN(INITIAL);
               }
<N1>.          {/* Bad word. Error. */
                yymore();
                BEGIN E;
               }
```

La terminación `/{sep}` constituye un *contexto de cola*[18]. Se utiliza para indicar que el patrón de esa regla deberá estar segido por un separador, pero el carácter separador no es incluido en `yytext` y será reconocido en patrones de reglas posteriores, pero no en la actual. Se utiliza en estos patrones para evitar reconocer una palabra incompleta.

La expresión `""` se utiliza para indicar un *carácter vacío*, puesto que las palabras de $N1$ no añaden ningún carácter para formar el plural.

El patrón `.` empareja con cualquier carácter excepto con un avance de línea. Se utiliza para detectar palabras erróneas o aquellas que no han sido incluidas en el lexical. En las acciones de esta regla se utiliza `yymore` para que al activar el modo pánico, en `yytext` se almacene todo el texto no reconocido como válido. La regla del error es la siguiente:

```
<E>{nosep}*/{sep} {/* Panic mode. Recognise every character
                     * until the next separator.              */
                   printf(", ERROR --> %s", yytext);
                   BEGIN(INITIAL);
                  }
```

La acción `BEGIN(INITIAL)` hace que el reconocedor vuelva a las reglas que no tienen condición de arranque[19].

Por último las siguientes reglas permiten tratar los separadores.

```
""         |
\t         |
\n         {/* separator non significatives */
            ;
           }
","      |
";"      |
```

[18] *Trailing context.*

[19] Realmente es como si existiese una condición de arranque implícita llamada `INITIAL` en todas aquellas reglas que caracen de condición de arranque.

```
":"     |
"."     {/* Puntuation marks */
          printf(" Puntuation mark -> %s", yytext);
          BEGIN(INITIAL);
         }
```

Los espacios, tabuladores y caracteres de nueva línea se ignoran, pero las marcas de puntuación se reconocen como un componente léxico, ya que son significativas en el proceso de análisis sintáctico para detectar los límites de las oraciones.

En este ejemplo sólo se muestra por pantalla el género y el número. En el analizador real, se almacenan los datos en una estructura que se pasa al analizador sintáctico. El modo en que se enlazan los analizadores léxico y sintáctico se muestra en el capítulo 6.

Un aspecto importante que hay que tener en cuenta es el de la definición de `yytext`. La variable `yytext` está definida por defecto en Flex como un puntero a `char`. La ventaja de usar esta defincón es que no se producirán *overflows* en el buffer cuando se reconozcan componentes léxicos muy largos[20], ya que Flex va asignando memoria a `yytext` dinámicamente[21]. La desventaja es que no se puede modificar el contenido de `yytext`, ya que los resultados que se obtendrían serían impredecibles. Esto último implica que no se pueden utilizar funciones como `input` y `unput`.

Se puede definir la variables `yytext` como un array mediante la utilización de `%array` en la sección de declaraciones del programa Flex. El tamaño viene determinado por el valor de `YYLMAX`, originalmente 8 Kilobytes. Utilizando esta definicón se puede modificar el valor de `yytext` sin problemas. El único inconveniente que podría surgir sería el derivado del límite de tamaño impuesto por el array. En nuestro caso no es ningún problema, ya que no existen palabras de miles de caracteres[22].

5.3 El comportamiento no determinista

En los lenguajes naturales existen ambigüedades a todos los niveles: léxico, sintáctico y semántico. En este capítulo nos interesa ver el modo en que se pueden tratar las ambigüedades del nivel léxico. Algunos autores proponen utilizar un enfoque de ventana deslizante, generalmente de tamaño reducido (sobre tres palabras), empleando técnica estadísticas como las cadenas de Markov o los autómatas probabilísticos para determinar el componente léxico más probable que se corresponde con una palabra dada. Este enfoque tiene varios inconvenientes serios:

- Es preciso disponer de información estadística actualizada acerca de las probabilidades de aparición conjunta de un conjunto de categorías de palabras[23]. Es necesario disponer de un texto amplio ya etiquetado para poder entrenar al modelo. De la bondad del conjunto de entrenamiento dependerá en buena medida la corrección de los resultados obtenidos.

[20] Por ejemplo, bloques de comentarios de miles de caracteres.

[21] Cuando **`yytext`** cambia de tamaño es necesario volver a reconocer todos los caracteres del componente léxico, lo cual puede provocar una degradación importante del rendimiento para componentes léxicos muy grandes.

[22] Si se desea, se puede cambiar el tamaño por defecto de **`yytext`** utilizando un **`#define`** para establecer **`YYLMAX`** al valor deseado.

[23] Por ejemplo, la probabilidad de que un sutantivo esté seguido de un verbo y un adverbio, etc.

- Se está tratando de incorporar a nivel léxico información correspondiente a nivel sintáctico y semántico. Esto provoca una confusión entre las tareas realizadas en cada uno de dichos niveles. Además, debido a la imposibilidad de aplicar modelos matemáticos complejos a conjuntos grandes de palabras, se están realizando suposiciones sobre ventanas muy pequeñas que no abarcan, en la mayoría de los casos, estructuras sintácticas completas, por lo que la posibilidad de error es grande.

- La carga computacional asociada es muy elevada, lo que disminuye el rendimiento. Para mantener un rendimiento aceptable se precisa ejecutar el programa en máquinas muy potentes.

- Muchas veces estos modelos trabajan en conjunción con bases de datos, lo que degrada todavía más el rendimiento, ignorando además el conocimiento disponible acerca de la formación de las palabras a nivel léxico.

El enfoque que hemos adoptado ha sido el de ampliar las capacidades del reconocedor generado por Flex, incorporándole un comportamiento no determinista.

5.3.1 El comportamiento determinista de los reconocedores Flex

Los reconocedores generados por FLEX son deterministas. Esto quiere decir que una palabra es analizada únicamente una sóla vez y las acciones de las reglas disparadas se llevan a cabo una sóla vez. Sin embargo, mediante la utilización de la acción `REJECT`[24] se puede indicar al reconocedor que rechace la entrada reconocida en los patrones de las reglas de la condición de arranque actual y que seleccione la segunda mejor regla cuyo patrón coincida con la entrada[25]. Aunque el propósito para el que fue creada esta acción se refiere fundamentalmente a la recuperación de errores, su utilización permite conseguir un comportamiento no determinista local en la condición de arranque en la que se ejecutó[26]

Sin embargo, mediante la utilización de `REJECT` no es posible ir atrás en el análisis más que hasta el comienzo de la condición de arranque actual. Si el reconocedor pasa por varias de estas condiciones de arranque hasta llegar al final de la palabra, como es nuestro caso, no es posible ir retrocediendo paso a paso hasta la condición inicial.

Puestos en contacto con Vern Paxon[27], creador de Flex, nos confirma que, efectivamente, el comportamiento no determinista que se consigue mediante la utilización de la acción `REJECT` es sólo local, pues Flex tan sólo almacena información de la condición de arranque en la que se encuentra en cada momento. Nos sugiere que para conseguir un comportamiento no determinista global probemos a utilizar, al final del reconocimiento de cada palabra, una llamada a `yyless(0)` e inmediatamente a continuación una llamada a `BEGIN(INITIAL)`. Con ello se consigue vaciar el contenido de `yytext` y enviar al reconocedor a la condición inicial, denominada `INITIAL`. Sin embargo, esto provocaba la entrada del reconocedor en un bucle infinito, puesto que al carecer de información global sobre el reconocimiento, el analizador léxico no *recordaba* las reglas de la condición

[24] Esta acción fue incorporada en la versión 2.1 de Junio de 1989.

[25] Recordemos que para Flex la mejor regla (la *mejor elección*) es aquella cuyo patrón coincide con la mayor cantidad de caracteres de la entrada. En caso de que dos patrones *empaten*, se considera mejor el situado antes en el código fuente.

[26] Un aspecto importante a tener en cuenta es que la acción **`REJECT`** actúa como una ramificación, de modo que el código situado después de ella en las acciones de las reglas no es ejecutado.

[27] Se puede contactar con Vern Paxon mediante e-mail en la dirección **`vern@ee.lbl.gov`**.

de arranque inicial que ya habían sido activadas, por lo que siempre lanzaba una y otra vez la mejor regla de la condición inicial.

5.3.2 La incorporación del no determinismo a los reconocedores Flex

Puesto que la carencia del no determinismo en los reconocedores generados por Flex vienen dada porque no se almacena información global sobre el texto reconocido, la solución consiste en encontrar algún método de incorporar a los reconocedores tal información. A priori, ésta puede parecer una tarea desbordante, ya que la consecución de no determinismo en todas las condiciones de arranque significaría tener que guardar una estructura arborescente, puesto que habría múltiples caminos mediante los cuales reconocer una palabra.

Sin embargo, si enfocamos bien el problema, se puede observar que éste radica en la iteración continuada en las reglas iniciales. Por lo tanto, de lo que se trata es de buscar un mecanismo que indique al reconocedor que un patrón ya ha sido reconocido para evitar que vuelva a lanzar la regla correspondiente.

De este modo, mediante la combinación de `yyless(0); BEGIN(INITIAL)` y esa información del *matching* de los patrones se puede conseguir un comportamiento no determinista del autómata reconocedor.

La solución adoptada se ha mostrado simple y eficiente, pues tan sólo es necesario mantener las dos variables siguientes:

- `match_no`, que indica el número de patrones que han sido emparejados con éxito hasta el momento en el proceso de reconocimiento de la palabra actual. Por tanto también indica el orden que ocupa en cuanto a *mejor elección* en términos de Flex.
- `semaphore` se utiliza para determinar cuándo se puede reconocer un determinado patrón. Su función es la de actuar de semáforo:
  - Si su valor es 0, el patrón que intentaba reconocer el analizador léxico es un patrón válido, puesto que no se había utilizado antes para intentar reconocer la palabra actual.
  - Si su valor es mayor que cero, indica la distancia al patrón correcto, en cuanto a *mejor elección* en términos de Flex, en relación al patrón que se está intentando reconocer.

Inicialmente ambas variables establecen sus valores a 0. Cuando se reconoce un patrón para una palabra, se incrementa en 1 el valor de `match_no` y si el valor de `semaphore` es 0, se iguala el valor de ésta al de `match_no`. Cuando se termine de reconocer esa palabra o se deseche por ser errónea, se llama a `yyless(0)` y `BEGIN(INITIAL)`, con lo cual el analizador léxico intentará reconocer la palabra otra vez desde el principio.

El reconocedor intentará entrar otra vez por el mismo patrón de antes, pero esta vez `semaphore` tendrá un valor distinto de 0 (concretamente tendrá valor 1), lo que indica que no debemos escoger ese patrón, sino otro que es la *siguiente mejor elección*. Se decrementa su valor en 1 y se hace un `REJECT`, con lo cual se busca precisamente la siguiente mejor elección.

El siguiente patrón reconocido será el corecto puesto que `semaphore` tendrá valor 0. Se incrementa el valor de `match_no` y se prosigue con el proceso.

Se debe de contemplar la posibilidad de una salida para el caso de que ya no haya más patrones correctos para la palabra examinada. Para ello se llama a `yymore()` y a `BEGIN S` desde una regla con el punto como patrón. La condición de arranque `S` contiene la regla siguiente:

```
/* Success condition */
<S>{nosep}*/{sep} {
                   BEGIN 0;
                   return (1);
                  }
```

La acción `BEGIN 0` es equivalente a `BEGIN(INITIAL)`. Esta regla es la única en la que se realiza un `return`, por lo que una vez que se llama a la función `yylex` desde el analizador sintáctico, no se devuelve el control hasta que se llega a la condición `S`, hecho que se da cuando ya se han examinado todos los posibles componentes léxicos válidos para una palabra.

Existe también una condición de error, llamada `E`, a la que se manda el reconocedor cuando una palabra no puede ser construida mediante el camino examinado:

```
/* Error condition */
<E>{nosep}*/{sep}  {
                    yyless(0);
                    BEGIN(INITIAL);
                   }
```

Es igual que la regla con <S> excepto que ésta no devuelve el control al analizador sintáctico, sino que intentará encontrar otro camino para reconocer la palabra.

5.3.3 Un pequeño ejemplo

Para facilitar la compresión de la incorporación del comportamiento no determinista, vamos a utilizar un pequeño ejemplo. Como en castellano hay un gran número de reglas léxicas, lo que nos obligaría a definir numerosas condiciones de arranque para crear un ejemplo mínimamente real, hemos sonsiderado más conveniente utilizar como ejemplo un pequeño analizador léxico en el que se muestra cómo reconocer todas las variantes de la palabra inglesa `bellow`. Este ha sido el ejemplo sobre el que hemos trabajado con Vern Paxon. La palabra `bellow` tienen cuatro entradas en un diccionario[28]:

bellow[1] *noun* bramido, rugido.

bellow[2] *intransitive verb* gritar, vociferar.

bellow[3] *transitive verb* bramar, rugir.

bellows[4] *noun plural* fuelle.

El problema surge al tener que reconocer una palabra como `bellows`[29], que puede ser:

- El singular del sustantivo `bellows`.
- El plural del sustantivo `bellow`.

[28] Estas acepciones han sida sacadas del *Diccionario Collins English-Spanish* editado por Grijalbo.

[29] El honor de encontrar este ejemplo ha sido del Prof. Jorge Graña. Su habilidad en el manejo de diccionarios es incuestionable.

- La tercera persona del presente del verbo `bellow`.

Para ser capaces de tratar con estas definiciones, debemos crear dos condiciones de arranque, que son las siguientes:

- `N` para los sustantivos.
- `V` para los verbos.

Para ello utilizamos la siguiente línea de código, en la que también incluimos la declaración de la condición `E` de error:

```
/* Start conditions */
%x E N V
```

A continiuación se definen los patrones de los separadores y de los no separadores:

```
/* Separators */
sep     [" ",;:\.\t\n]

/* Non separators */
nosep   [^" ",;:\.\t\n]
```

Algo muy importante es declarar las variables enteras `match_no` y `semaphore`:

```
%{
unsigned int match_no, semaphore;
%}
```

A continuación ya se pasa a escribir las reglas léxicas con el no determinismo incorporado, no sin antes haber escrito un pequeño trozo de código en el cual se establece a cero el valor de las dos variables anteriores. Esta porción de código situado después de `%%` y encerrada entre `%{` y `%}` se ejecutrá cada vez que se llame a la función `yylex`.

```
%%

%{
/* Code executed each time yylex is called */
match_no = 0;
semaphore = 0;
%}

/* Error condition: Panic mode */
<E>{nosep}*/{sep}    { printf("%s -> error \n\n", yytext);
                       BEGIN(INITIAL);RETURN(1);
                     }

/* Verbs */
<V>s/{sep}           printf("%s -> verb, 3rd        \n",yytext);REJECT;
<V>ed/{sep}          printf("%s -> verb, past       \n",yytext);REJECT;
<V>ing/{sep}         printf("%s -> verb, gerund     \n",yytext);REJECT;
<V>""/{sep}          printf("%s -> verb, infinitive \n",yytext);REJECT;
<V>""/.              yyless(0); BEGIN(INITIAL);

/* Nouns */
<N>s/{sep}           printf("%s -> noun, plural   \n",yytext);REJECT;
<N>""/{sep}          printf("%s -> noun, singular \n",yytext);REJECT;
```

```
<N>""/.             yyless(0);BEGIN(INITIAL);

/* WORDS */
bellow              { if ( semaphore!=0) { semaphore--; REJECT; };
                      match_no++; semaphore=match_no;

                      yymore(); BEGIN(V);
                    }

bellow              { if ( semaphore!=0) { semaphore--; REJECT; };
                      match_no++; semaphore=match_no;

                      yymore(); BEGIN(N);
                    }

bellows             { if ( semaphore!=0) { semaphore--; REJECT; };
                      match_no++; semaphore=match_no;

                      yymore(); BEGIN(N);
                    }

/* Separators */
" "                 |
\t                  |
\n                  |
","                 |
","                 |
";"                 |
":"                 |
"."                 printf("%s -> separator \n\n'', yytext);return(1);

.                   yymore(); BEGIN (E);

<<EOF>> yyterminate();

%%
```

Mediante la utilización de las dos líneas iniciales de código de todas las reglas con comportamiento no determinista

```
if ( semaphore!=0) { semaphore--; REJECT; }
match_no++;semaphore=match_no;
```

se comprueba si el valor del semáforo es correcto. Si no lo es, se realiza un `REJECT`, con lo cual el resto de las acciones de esa regla no se ejecutan. En caso contrario, se actualizan las variables convenientemente. Para facilitar la escritura de reglas en Flex, se puede definir una macro `NDFLEX` que contenga esas instrucciones, que van a ser iguales en todas las reglas que se quiere que participen del comportamiento no determinista.

La última regla permite evitar que el reconocedor genere errores cuando se intenta llamar a `yylex` y ya se ha alcanzado el fin de fichero. Puede ser eliminada y el programa seguirá funcionando, pero se recomienda que sea incluida.

En este ejemplo, por sencillez, no se ha utilizado la condición de arranque `S`, sino que `yylex` devuelve el control cuando encuentra un separador. En un programa real los

espacios, tabuladores y retornos de carro se ignorarían y las marcas de puntuación se reconocerían como componentes léxicos.

Aunque el ejemplo se ha puesto en inglés por cuestiones de legibilidad, en castellano se dan muchos más casos de ambigüedades. Por ejemplo, la palabra `para` puede ser reconocida como:

- Una preposición.
- La tercera persona del presente de indicativo del verbo "parar".
- La segunda persona del imperativo del verbo "parar".
- La primera persona del presente de subjuntivo del verbo "parir"
- La tercera persona del presente de subjuntivo del verbo "parir"

Existen muchos casos como éste, pero para mostrarlos es necesario escibir muchas reglas debido a la complejidad de la conjugación verbal, por lo que son poco adecuados para un ejemplo didáctico.

5.3.4 Conclusiones

Es posible incorporar nuevas capacidades a los reconocedores generados por una herramienta tan flexible como Flex. El enfoque utilizado es conceptualmente simple pero demuestra una gran potencia en las realizaciones prácticas. Mediante la utilización de macros puede ser incorporado fácilmente en cualquier programa.

Aunque aquí se ha usado sólo en reglas con un único patrón, este enfoque funciona perfectamente en reglas con múltiples patrones, puesto que al construir las tablas del autómata que subyace tras el reconocedor, el resultado es el mismo que si se hubiese asociado a cada patrón una regla distinta. Con la definición de reglas multi-patrón lo que se consigue es un menor consumo de memoria al evitar tener que definir múltiples veces el mismo código en el programa C que genera Flex.

Se puede utilizar este enfoque en las reglas de cualquier condición de arranque. Un modo de hacerlo manualmente consiste en definir un par de variables para cada condición, utilizando el nombre de la condición como prefijo o sufijo de los nombres de las variables, y definir las macros correspondientes. Dado que el proceso de creación de tales variables y macros es automatizable, sería deseable que en posteriores versiones de Flex se incluyese esta característica de modo automático, dejando al usuario la libertad de incluir las macros no deterministas en las reglas.

5.4 La integración con ICEeditor

Desde el punto de vista del usuario, no existe ningún tipo de interacción entre el analizador léxico y ICEeditor. Ciertamente, no existe ningún medio por el cual el usuario pueda actuar directamente sobre el reconocedor léxico. Es tarea del analizador sintáctico realizar las llamadas que sean necesarias a la función `yylex` para obtener los componentes léxicos del texto. El usuario dispone de medios para llamar al parser en modo incremental o de reanálisis total. La actividad del analizador léxico le es totalmente transparente.

Sin embargo, desde un punto de vista de integración de todos los elementos, sí es necesaria una comunicación directa entre ICEeditor y el lexical. Esto se debe a que ICEeditor precisa obtener información acerca de la situación de los componentes léxicos en la pantalla. Hablando más propiamente, necesita conocer dónde empieza y dónde termina el texto de cada componente léxico reconocido, tomando como referencia el texto total que se está analizando. Sin esta información sería imposible proporcionar servicios de edición de componentes léxicos.

Esta información no puede ser obtenida a través del analizador sintáctico, puesto que los analizadores generados por ICE ignoran cualquier información posicional a nivel de carácter, interesándose únicamente por la secuencia de componentes léxicos que han sido reconocidos por el analizador léxico.

Puesto que el analizador léxico sí que está directamente involucrado en el manejo de los caracteres, será con este módulo con el que ICEeditor tendrá que establecer la comunicación para recuperar la información que le interesa.

Como ICEeditor ha sido escrito en LE-LISP y el analizador léxico ha sido generado en código C, es necesario utilizar la interfaz de comunicación entre ambos lenguajes que se muestra en el capítulo 4. Concretamente, se van a utilizar las facilidades de enlace dinámico de los programas LE-LISP con módulos C compilados para establecer esta comunicación.

5.4.1 Las variables de control de posición

El analizador léxico, a medida que va reconociendo los caracteres, debe mantener un conjunto de variables que permitan establecer un control exhaustivo sobre la posición en la que se encuentra cada componente léxico en el texto que está siendo analizado. Para cada componente léxico, es necesario conocer la posición de comienzo y su longitud. En la posición se debe indicar tanto el número de línea como la posición dentro de esa línea. Conforme a esto, es presiso definir las siguiente variables[30]:

- `token_line`, que almacenará el número de la línea en la que comienza la palabra que está siendo actuamente sometida a análisis.

- `token_pos`, que guarda la posición, dentro de la línea indicada por la variables anterior, en la que comienza la palabra que está siendo analizada.

- `token_len`, que almacena la longitud del texto correspondiente al token reconocido.

La principal diferencia de tratamiento que van a recibir estas variables estriba en que las dos primeras establecen sus valores nada más comenzar a reconocerse una palabra, mientras que la tercera debe esperar al final del reconocimiento del componente léxico, para poder fijar su valor al de la longitud del texto almacenado en `yytext`.

[30] Todas las declaraciones de variables y tipos de datos, así como las declaraciones de funciones, deben realizarse en un fichero de cabecera (aquellos fichero C que tienen la extensión `.h`) que debe ser incluido en la porción del programa Flex reservada a la definición de código C dentro de la sección de declaraciones. La definición de las funciones debe realizarse también en un programa C separado que una vez compilado debe ser enlazado con el analizador léxico generado, utilizándose para ello la instrucción `ld` con el parámetro `-r`.

5.4.2 La lista de posiciones

Sin embargo, con las variables anteriores no es suficiente. La causa está en que una vez que ICEeditor cede el control al analizador sintáctico, éste no lo devuelve hasta que termina de analizar todo el texto (o la porción modificada incrementalmente, en su caso). Una forma de solucionar ésto es hacer que el analizador léxico vaya almacenando la posición de cada componente léxico en un lista enlazada, a la que ICEeditor podrá acceder una vez que el analizador sintáctico haya terminado su tarea.

Para ello hemos de definir los tipos de datos correspondientes y las funciones de manejo asociadas. El primer paso consiste en definir una estructura C en la cual se almacenará la información correspondiente a cada componente léxico individual. Esta estructurá recibirá el nombre de `ITEM_TOKEN_INFO`. Como se van a enlazar varios de estas estructuras en una lista, es conveniente definir un tipo `TOKEN_INFO` que sea un puntero a una de estas estructuras. Su función es hacer de puntero al primer elemento de la lista. Las definiciones en C se muestran a continuación:

```
typedef struct ITEM_TOKEN_INFO *TOKEN_INFO;
struct ITEM_TOKEN_INFO
{
    int token_line;
    int token_pos;
    int token_len;
    TOKEN_INFO_LIST next_item;
}
```

El siguiente paso consiste en definir un conjunto de funciones que permitan manejar cómodamente listas de tipo `TOKEN_INFO`. Utilizando una notación al estilo Lisp podemos definir las siguientes funciones:

- `reset_token_info`, cuya misión será la de vacíar la lista con la información posicional de los componentes léxicos. Para ello va recorriendo la lista y va liberando el espacio de memoria ocupado por cada elemento.

- `is_empty_token_info`, que informa de si una lista está vacía o no. Para ello es suficiente con determinar si el puntero de tipo `TOKEN_INFO` que apunta al primer elemento de la misma es igual a NULL.

- `nextl_token_info`, que elimina el primer elemento de la lista, liberando el espacio de memoria.

- `nconc1_token_list`, que añade un elemento al final de la lista. Para mejorar la eficiencia de esta función es aconsejable utilizar una variable global o estática que apunte al último elemento de la lista.

- `create_item_token_info_list`, es la función encargada de crear los elementos de tipo `ITEM_TOKEN_INFO` que forman parte de la lista de información de componentes léxicos.

Mediante el tipo `TOKEN_INFO` y las funciones asociadas disponemos de las herramientas necesarias para realizar el control de posición de los componentes léxicos en el reconocedor léxico.

Para facilitar el proceso de interacción entre LE-LISP y C, se definen las funciones siguientes, que permiten obtener por separado cada uno de los componentes del CAR de una lista con información posicional de los componentes léxicos:

- `ncar_token_line`, que obtiene el campo `token_line`.
- `ncar_token_pos`, que obtiene el campo `token_pos`.
- `ncar_token_len`, que obtiene el campo `token_len`.

Estas funciones no tienen argumentos, sino que acceden directamente a una variable global denominada `token_info_list`, toman el car de la lista a la que apunta mediante una llamada a `CAR_TOKEN_INFO` y recuperan el campo de interés, que será el resultado que devuelvan.

Una función adicional, llamada `nextl_token_info_list`, llama a `NEXTL_TOKEN_INFO` para eliminar el CAR de la lista de información. Comprueba si la lista resultante es vacía, en cuyo caso devuelve el valor 0. Si la lista no está vacía, devuelve el valor 1.

5.4.3 El control de posición en el analizador léxico

Para controlar la posición de cada componente léxico, mantenemos las cuatro variables globales:

- `token_info_list`, de tipo `TOKEN_INFO`, que apunta al inicio de la lista de información de componentes léxicos. Sólo se inicializa una vez.
- `item_token_info_list`, un puntero al elemento de la lista en el que se va almacenando la infomación del componente léxico que se está reconociendo. Se debe crear un nuevo elemento en cada palabra y asignarlo a este puntero.
- `last_token_info_list`, un puntero al último elemento de la lista. Se utiliza para aumentar la eficiencia del proceso de enlace de nuevos elementos al final de la lista.
- `line_no`, una variable entera en la que se almacena el número de línea actual en el que se encuentra el reconocedor.
- `pos_no`, en la que se almacena la posición dentro de la línea del carácter en el que se encuentra el proceso de reconocimiento de palabras.

El procedimiento que se sigue para el control de la posición se resume a continuación:

- Se incializan a cero las variables `line_no` y `pos_no`.
- En la última regla que se ejecuta en la detección no determinista de cada palabra[31], se crea un elemento de tipo `ITEM_TOKEN_INFO` mediante una llamada a la función `malloc`, se asigna a la variable `item_token_info_list` y se rellenan los campos `token_line` y `token_pos` con los valores de `line_no` y `pos_no`, respectivamente.

[31] La regla sin condición de arranque cuyo patrón está formado únicamente por el punto y cuyas acciones envían al reconocedor a la condición de arranque <s>.

- Cuando se llega a la regla de la condición de arranque <S>, se rellena el campo `token_len` de la estructura apuntada por `item_token_info` con el valor obtenido de calcular la longitud del texto almacenado en `yytext`. Este valor también se suma a `pos_no` para actualizar el valor de esta variable. Se enlaza la información del nuevo componente léxico a la lista mediante una llamada a la función `CONC1_TOKEN_INFO`.
- La regla que trata con los caracteres separadores no significativos debe ser partida en tres: una que trate con espacios en blanco, otra con tabuladores y otra para los caracteres de nueva línea:
  - En la regla del tabulador y la del espacio tan sólo hay que actualizar convenientemente el valor de `pos_no`.
  - En la regla de los caracteres de nueva línea, se debe incrementar el valor de `line_no` y establecer a cero el valor de `pos_no`.

Las reglas que tratan con los caracteres de puntuación se tratan igual que las que reconocen lexemas, pues en este analizador, una marca de puntuación es equivalente al lexema de un componente léxico, cuyo texto es el propio carácter de puntuación.

5.4.4 La recuperación de información de posición por ICEeditor

Para que ICEeditor pueda acceder a las funciones definidas en el módulo del analizador léxico, es preciso realizar una llamada a la función `cload` con el nombre del módulo compilado de dicho analizador como argumento.

También es preciso definir como externos los símbolos y las funciones a las que se va a acceder. Para ello se utiliza la función `defextern`:

```
(defextern _car_token_line () fix)
(defextern _car_token_pos  () fix)
(defextern _car_token_len  () fix)
(defextern _nextl_token_info_list () fix)
```

Tal como se indica en el capítulo 4, el primer parámetro de `defextern` indica el nombre de la función en C a la que se desea acceder, precedida de un guión bajo. El segundo indica el tipo de los parámetros que se le pasan a dicha función, en nuestro caso ninguno, y el tercero el tipo del valor devuelto, que en nuestro caso es siempre `fix`[32].

Con estos valores se puede acceder al árbol de enlace entre los componentes léxicos y el texto para actualizarlo convenientemente, mediante los métodos ya disponibles en ICEeditor.

Mediante `nextl_info_token_list` se borra el CAR de la lista de información de la posición de los componentes léxicos. Si el resultado devuelto es un 0, la lista resultante estará vacía y por lo tanto se habrá terminado con los componentes léxicos reconocidos en el proceso de análisis.

[32] Un número entero.

Capítulo 6

El análisis sintáctico

En este capítulo se tratan los aspectos correspondientes al analizador sintáctico. La discusión se centrará, más que en la escritura de gramáticas, en la utilización de la herramienta de generación de analizadores sintácticos no deterministas incrementales y su integración con el analizador léxico y con ICEeditor.

6.1 Introducción

La herramienta utilizada para la construcción del analizador sintáctico es ICE[1]. Esta herramienta es capaz, a partir de una gramática de contexto libre escrita en el mismo formalismo que el utilizado por Yacc [Johnson 75], de generar un analizador no determinista e incremental para dicha gramática. En el apéndice A se muestra una visión general de los fundamentos teóricos de ICE. El presente capítulo se centrará en los aspectos prácticos, es decir, en la utilización del conjunto de programas que implemetan las ideas mostradas en dicho apéndice.

Para conseguir una compatibilidad total con Yacc, Vilares [Vilares 92] tomó como punto de partida el código de Bison [Donnelly y Stallman 88]. Ésta es una herramienrta de generación de analizadores sintácticos de contexto libre, compatible con Yacc y desarrollada por Robert Corbett y Richard Stallman. Una diferencia muy importante con respecto a Yacc es que el copyright pertenece a la Free Software Foundation y el producto se encuentra sometido a la "GNU General Public License", lo cual implica que se puede acceder al código fuente y que éste puede ser modificado y distribuido ateniéndose a los términos y condiciones de la licencia GNU. En cambio, el código de Yacc es propietario y no puede ser modificado libremente.

La herramienta ICE en sí está construida mediante un conjunto de módulos C, ficheros de código y de cabecera, que al ser compilados y enlazados producen un ejecutable de nombre `bison`. Los ficheros C son variaciones de los originales existentes en la distribución de Bison. Las modificaciones se refieren a los cambios sustanciales que fue preciso introducir para que los analizadores generados exhibiesen un comportamiento no determinista y tuviesen capacidades incrementales.

Relacionado con lo anterior, desde el punto de vista del usuario se ha producido también un cambio importante: el analizador generado a partir de la gramática de entrada no tiene

[1] **I**ncremental **C**ontext-free **E**nvironment.

ya la forma de un programa C, sino la de un programa LE-LISP.

Cualquier programa LE-LISP puede ser enlazado dinámicamente a módulos C compilados y el sistema LE-LISP puede ser invocado desde un programa escrito en C[2]. Por tanto, no se pierde potencia en cuanto a la capacidad de integrar el analizador con otras herramientas, como por ejemplo los analizadores léxicos generados por Flex, ni de ser utilizado en otros programas, como por ejemplo una aplicación con entrada salida interactiva que realice el parsing de los datos de entrada utilizando funciones generadas por Bison.

La utilización de un analizador sintáctico escrito en LE-LISP proporciona una serie de ventajas, como son:

- Disponibilidad de un entorno interactivo para que el usuario pueda entablar una mejor comunicación con el analizador. Esto hace necesaria la existencia de la parte interpretativa de ICE[3]. Esta parte proporciona medios al usuario para que sea capaz de realizar un seguimiento de la actividad del analizador, incluyendo capacidades de trazado y depuración.

- Facilidad de integración con la variedad de sistemas existentes en el entorno LE-LISP, que van desde complejos sistemas orientados al desarrollo de lenguajes, como CENTAUR [Centaur 92a], hasta sofisticadas herramientas de construcción de interfaes gráficos en entornos de ventanas, como son MASAÏ [ILOG 92g] y AÏDA [ILOG 92c].

6.2 Utilización de ICE

En esta sección se van a tratar los temas relacionados con la utilización correcta de la herramienta ICE. Se va a mostrar cómo se deben utilizar los ficheros de configuración, cómo trabajar con los ficheros de entrada y cual es la finalidad de los ficheros de salida.

6.2.1 Ficheros de configuración

El primer paso para trabajar correctamente con ICE es establecer adecuadamente el contenido de los ficheros de configuración. Estos ficheros establecen ciertas características del comportamiento del analizador generado. No están relacionados con la configuración del generador ICE.

Existen dos ficheros involucrados en la configuración, que reciben los nombres de `.lelisp` y `.ice`.

El fichero `.lelisp`

Este fichero es realmente un fichero de configuración de LE-LISP, en el cual el usuario puede introducir cualquier expresión válida en este leguaje. Este fichero, que debe estar ubicado en el directorio *home* del usuario, es leido durante el arranque del sistema LE-LISP, por lo que se utiliza para incorporar al sistema definiciones de funciones que van a

[2] En el capítulo 4 se muestran los tipos de enlaces existente entre los lenguajes LE-LISP y C y cómo se pueden realizar.

[3] En Bison [Donnelly y Stallman 88] sólo existe la parte de generación de analizadores, los cuales son totalmente autónomos.

ser utilizadas posteriormente con asiduidad y para llamar a funciones de inicialización y carga de módulos.

En lo que respecta a ICE, la utilización de este fichero se restringe en la mayoría de los casos a la definición de una función denominada `init-ice`, cuya finalidad es:

- Cargar los ficheros con la definiciones de las estructuras de datos LE-LISP utilizadas en el analizador generado. Estos ficheros tienen la extensión `.lh`.
- Cargar la librería circular estándar `libcir.ll`.
- Cargar las ficheros que contienen el código LE-LISP del módulo de ICE encargado de interpretar el analizador generado.

Un ejemplo típico del contenido de este fichero, en lo referente a ICE, es el que se muestra a continuación

```
/* ICE initialization */
(defun init-ice ()
        (load "/home/galena/ice/interpret/classic/change.lh")
        (load "/home/galena/ice/interpret/classic/ice.lh")
        (load "/home/galena/ice/interpret/classic/transition.lh")
        (load "/home/galena/ice/interpret/classic/order.lh")
        (load "/home/galena/ice/interpret/classic/item.lh")
        (load "/home/galena/ice/interpret/classic/itemset.lh")

        (libload "libcir.ll")

        (load "/home/galena/ice/interpret/classic/gc.ll")
        (load "/home/galena/ice/interpret/classic/order.ll")
        (load "/home/galena/ice/interpret/classic/item.ll")
        (load "/home/galena/ice/interpret/classic/itemset.ll")
        (load "/home/galena/ice/interpret/classic/ice.ll")
        (load "/home/galena/ice/interpret/classic/tools.ll"))
```

El fichero `.ice`

El fichero propio de configuración de ICE recibe el nombre de `.ice` y debe estar ubicado en el directorio *home* del usuario. Se utiliza principalmente para indicar la localización de los ficheros correspondientes al código del analizador sintáctico (código LE-LISP generado por ICE) y del analizador léxico (código C, generalmente generado por Lex o Flex[4]). Para ello se utiliza la función `{ice}:user-language`, que toma como argumentos dos cadenas de carácteres, el nombre del lenguaje (el mismo que se utiliza al llamar a la función `{ice}:load`) y el directorio en el cual se encuentran los ficheros mencionados anteriormente.

En la siguiente porción de código se muestra el contenido de un fichero `.ice` en el que se indica dónde buscar lenguajes como los del futuro sistema GALENA[5], Pascal, de expresiones aritméticas, etc.

[4] No es obligatorio utilizar Flex ni Lex para construir el lexical. Es suficiente con tener un programa C compilado en el que esté definida la función `yylex`.

[5] GALENA, de **G**enerador de **A**nalizadores para **LE**nguajes **NA**turales, es el nombre de un proyecto en desarrollo cuyo fin es el de construir herramientas para el análisis de lenguajes naturales, abarcando los aspectos léxicos, sintácticos y semánticos.

```
({ice}:user-language "galena" "/home/galena/galena/tables/work/")
({ice}:user-language "pascal" "/home/cialonso/ice/tables/pascal/")
({ice}:user-language "pascal_nd" "/home/cialonso/ice/tables/pascal_nd/")
({ice}:user-language "nogo" "/home/cialonso/ice/tables/nogo/")
({ice}:user-language "arit" "/home/cialonso/ice/tables/arit/")
({ice}:user-language "arit_d" "/home/cialonso/ice/tables/arit_d/")
({ice}:user-language "lambda" "/home/cialonso/ice/tables/lambda/")
({ice}:user-language "brackets" "/home/cialonso/ice/tables/brackets/")
({ice}:user-language "metal" "/home/cialonso/ice/tables/metal/")
```

6.2.2 Ficheros de entrada

Para que ICE pueda generar un analizador sintáctico, necesita un fichero que contenga la descripción de una gramática de contexto libre a partir de la cual poder construir dicho analizador. Como ya se comentó anteriormente, el formato de esta gramática es el mismo que el utilizado por Bison y Yacc. Este tipo de ficheros tiene la extensión `.y`. Para una descripción completa del formato del fichero, puede acudirse a [Donnelly y Stallman 88]. Para comprender mejor la forma en que interactúan Yacc y Lex[6], una buena referencia es [Mason y Brown 90]. En [Aho et al. 90] y [Aho y Ullman 73] se realiza un detallado estudio de las gramáticas de contexto libre aplicadas a lenguajes de programación. Para un estudio general de este tipo de gramáticas puede recurrirse a libros como [Sudkamp 88], [Hopcroft y Ullman 79] y [Harrison 87].

Una vez activado el intérprete de ICE, el usuario puede solicitar que sea cargado el analizador para un lenguaje *lang*. En tal caso, ICE busca, en el directorio indicado en la llamada a la función `{ice}:user-language` en la cual se declaró la existencia del lenguaje *lang*, los siguientes ficheros:

- `yacc`*lang*`.tab.c`, que contiene el código LE-LISP del analizador sintáctico para el lenguaje *lang*.
- `lex`*lang*, que contiene el ejecutable del analizador léxico. Para obtener este ejecutable se deben compilar los distintos módulos[7] con la opción `-c` del compilador C y enlazarlos mediante `ld` utilizando la opción `-r`. Si se ha utilizado Lex o Flex para para construir el analizador léxico, es necesario indicar al editor de enlaces que debe utilizar la librería `l` mediante la opción `-ll`.

6.2.3 Ficheros de salida

El módulo generador de ICE, cuyo ejecutable mantiene el nombre de `bison`, construye a partir de la gramática contenida en `yacc`*lang*`.y` el código LE-LISP del analizador sintáctico para el lenguaje *lang*. Para ello debe especificarse la opción `-n`[8] . Todo el código generado se almacena en el fichero `yacc`*lang*`.tab.c`. La terminación `.c` de este fichero se mantiene

[6] O sus equivalentes GNU, Bison y Flex.

[7] Si se ha utilizado Flex o Lex, el módulo principal (aquel que contiene las rutinas propias del análisis léxico, incluyendo la función **yylex**) será **lex.yy.c** y los demás módulos contendrán rutinas definidas por el usuario.

[8] En el resto del texto se supondrá, aunque no se indique explícitamente, que siempre que se llama a **bison** se utiliza la opción **-n**. Esta opción le indica a **bison** que debe generar un analizador sintáctico no determinista incremental en LE-LISP, es decir, un analizador ICE. Si no se especifica la opción **-n**, el analizador generado será de tipo determinista tradicional escrito en C, esto es, un analizador típico de Bison o Yacc.

por cuestiones de compatibilidad con los nombres de los ficheros generador por Bison. Ciertamente, resulta curioso que un fichero con la extensión `.c` contenga código Lisp.

Si se especifica la opción `-d` en la llamada a `bison`, se generará un fichero `yacc`*lang*`.tab.h` que, como su extensión indica (esta vez sí), es un fichero de cabecera para el lenguaje C. Contiene la definición de los valores enteros que identifican a cada comonente léxico, por lo que debe ser utilizado en el programa que implementa el analizador léxico. Para ello se utiliza una directiva `#include` en el código fuente del lexical. En los programas Flex, esta declaración se incluye en la zona de código C de la sección de definiciones[9]. Puesto que el fichero objeto del analizador léxico dependerá de este fichero de cabecera, debe tenerse cuidado de llamar primero a `bison` y depués a `lex` o `flex`, ya que de lo contrario el lexical puede estar utilizando definiciones de componentes léxicos incongruentes con las del parser.

6.3 Integración con el analizador léxico

Los analizadores sintácticos generados por ICE se integran perfectamente con los analizadores léxicos generados mediante Lex o Flex, al igual que lo hacían sus antecesores Yacc y Bison. Esta integración se basa fundamentalmente en:

- El valor devuelto por las llamadas a la función `yylex`, que es un entero representando el comonente léxico reconocido[10], excepto el valor 0, que representa la condición de fin de fichero.
- El valor semántico de cada componente léxico, almacenado en la variable `yylval`.

El problema surge cuando queremos incorporar el no determinismo al reconocimiento léxico de los componentes léxicos. La función `yylex` sólo es capaz de devolver un valor entero. Cuando se trata con reconocimiento no determinista, el analizador sintáctico debe recibir una lista de componentes léxicos por cada palabra que ha sido reconocida por el lexical.

6.3.1 La variable `token`

Para facilitar la interacción del analizador sintáctico son el reconocedor léxico no determinista, se ha optado por mantener intactas las estructuras generadas por Flex y añadir una nueva variable, denominada `token`, que almacene la lista de componentes léxicos reconocidos en cada ejecución de la función `yylex`.

La variable `token` tiene como tipo una nueva estructura denominada `ice_lex_object`. Esta estructura, que se define en el fichero de cabecera que contiene las declaraciones de tipos y fuenciones C utilizadas en el reconocedor generado por Flex, se muestra en el siguiente fragmento de código:

```
struct ice_lex_object
{
        char          *word;
        INT_LIST      category,
```

[9] Es decir, entre los delimitadores `%{` y `%}` existentes antes del primer `%%`.

[10] Este entero se identifica con los componentes léxicos mediante los valores indicados en los `#define` incluidos en el fichero `.tab.h`. Tales valores se generan a partir de las definiciones `%token` incluidas en el programa fuente (con extensión `.y`).

```
                    gender,
                    number,
                    person,
                    verbal_tense;
        STRING_LIST lemma;
}
```

El siginificado de los distintos campos es el siguiente:

- `word`. Este campo almacena la cadena de caracteres correspondiente a la palabra que ha sido reconocida en la última llamada a la función `yylex`.
- `category`. Es una lista de valores enteros que almacena el tipo de componente léxico que se ha reconocido. Cada uno de los valores se corresponde con una declaración `%token` en el fichero de la gramática. Para comprender mejor su significado, diremos que cada valor de esta lista se corresponde con un valor válido que podría ser devuelto por la función `yylex` cuando se trata de reconocedores léxicos deterministas convencionales.

Estos dos campos cosnstituyen la información básica que se debe almacenar en `token` para permitir el enlace entre los anlizadores léxico y sintáctico. Por tanto, debe ser incorporada en cualquier programa Flex que pretenda utilizar el reconocimiento no determinista de los componentes léxicos. El resto de la información almacenada en la variable `token` es específica de la aplicación que se trata en este trabajo. Diferentes aplicaciones pueden redefinir la estructura `ice_lex_object` para adaptar el contenido de los campos a sus necesidades específicas. Otra posible solución hubiese sido situar esta información dependiente de la aplicación en la variable `yylval`. Para ello habría que definir dicha variable como un puntero a una lista de estructuras en las que se almacenaría tal información. El problema surge al tratar de mantener la consistencia entre la lista de componentes léxicos (representada como una lista de `ice_lex_object`'s) y la lista de `yylval`'s. Para facilitar el mantenimiento de esta consistencia se ha considerado más adecuado almacenar toda la información que tiene relación con las capacidades no deterministas en una única estructura. Con ello se consigue adicionalmente un alto grado de aislamiento de estas nuevas características con respecto a las capacidades estándar de los reconocedores generados por Flex.

Los campos con información adicional dependiente de la aplicación que se han utilizado son:

- `gender`. Es una lista de enteros que almacena el código que identifica el género de aquellas palabras en las que es aplicable esta característica. Para aquellos componentes léxicos en los que no es aplicable, recibe el entero que se corresponde con el valor No-Aplicable. Esta lista está coordinada con la lista `category`, lo que quiere decir que el primer elemento de la lista de género indica el género del primer token en la lista de categorías, el segundo elemento en la lista de géneros el valor del género para el segundo componente léxico, etc.
- `number`. Este campo es una lista de enteros que indica el valor del número para aquellas palabras en las cuales es aplicable. Por tanto, es equivalente a la lista `gender`, salvo que los valores conciernen al número en vez de al género.

- `person`. Al igual que los campos anteriores, es una lista de enteros coordinada con la lista de categorías, pero que en este caso indica la persona (aplicable en el caso de verbos y pronombres).

- `verbal_tense`. Lista de enteros que indica el tiempo verbal de una palabra, si es aplicable.

- `lemma`. Este campo es una lista de cadenas de carácteres que indica el lema de cada palabra según cada tipo de componente léxico que ha sido reconocido para dicha palabra. Por ejemplo, tomemos la palabra `para`, que es reconocida como:

  - una preposición, en cuyo caso el lema es `para`.
  - Dos formas verbales del verbo parar[11], en cuyo caso el lema es `parar` para ambas. Cada forma verbal se corresponde con un componente léxico distinto y por lo tanto con dos elementos en cada una de las lista presentes en la estructura `token`.
  - Dos formas verbales del verbo parir[12], en cuyo caso el lema es `parir` para ambas.

6.3.2 Estructuras de datos auxiliares

Como se ha podido observar en la definicón de la estructura `ice_lex_object`, los campos que son listas de enteros tienen asignado el tipo `INT_LIST`, y los que son listas de cadenas de carácteres el tipo `STRING_LIST`. Estos dos tipos han sido definidos para facilitar la tarea de manejo de tales listas, ya que además de definir el tipo en sí se define un conjunto de funciones asociadas que permiten trabajar de un modo seguro y estandarizado sobre estas listas.

La lista de enteros

El tipo `INT_LIST` implementa las listas de enteros. La declaración de este tipo se realiza como se muestra en las siguientes líneas de código:

```
typedef struct ITEM_INT_LIST *INT_LIST

struct ITEM_INT_LIST
{
   int         data_item;
   INT_LIST    next_item;
};
```

Por tanto, cuando se declara una variable de tipo `INT_LIST`, se está declarando realmente un puntero a un objeto en memoria de tipo `ITEM_INT_LIST`, el cual es una estructura que almacena un valor entero, identificado como el campo `data_item` y un puntero al siguiente elemento de la lista, identificado mediante el campo `next_item`.

Para manejar este tipo de listas, se han definido las siguientes funciones:

[11] La tercera persona del presente de indicativo y la segunda del imperativo.
[12] La primera y la tercera persona del singular del presente de subjuntivo.

- `RESET_INT_LIST`. Esta función toma como argumento un puntero a un valor de tipo `INT_LIST`[13] y se encarga de eliminar una lista de enteros, liberando todas las posiciones de memoria ocupadas por todos sus elementos y estableciendo el puntero de inicio de la lista al valor `NULL`.

- `IS_EMPTY_INT_LIST`. Esta función comprueba si el valor que se le pasa como argumento es el inicio de una lista de enteros o no. El valor devuelto es un entero con significado booleano.

- `CONS_INT_LIST`. Esta función, al igual que la función `cons` del Lisp, añade un elemento en la cabeza de una lista de enteros. Para ello toma como argumento un puntero a un elemento `INT_LIST` y un valor entero, que será asignado al campo `data_item` de la estructura `ITEM_INT_LIST` que representa al nuevo elemento de la lista.

- `NEXTL_INT_LIST`. Esta función realiza un trabajo similar al de su homónima `nextl` en Lisp: elimina el primer elemento de una lista de enteros y devuelve el resto de la lista (En Lisp el CDR). Al igual que la función Lisp, esta función modifica físicamente sus argumentos, ya que no crea una lista nueva, sino que desengancha realmente el primer elemnto de la lista argumento[14].

- `CAR_INT_LIST`. Esta función obtiene, utilizando terminología Lisp, el CAR de una lista de enteros. Al igual que la función Lisp, sólo realiza operaciones de lectura, por lo que no modifica la lista argumento. En este caso el argumento es realmente un valor de tipo `INT_LIST`.

Con estas funciones se dispone de un conjunto lo suficientemente amplio de operadores para tratar las listas de enteros sin necesidad de recurrir a manipulaciones manuales de sus elementos.

La lista de cadenas de carácteres

Las listas de cadenas de carácteres, que por ahora sólo se utilizan para tratar el campo `lemma` de la variable componente léxico, se implementan mediante la utilización del tipo `STRING_LIST`, el cual se declara tal y como se muestra en las siguientes líneas de código:

```
typedef struct ITEM_STRING_LIST *SRING_LIST;

struct ITEM_STRING_LIST
{
   char *data_item;
   STRING_LIST next_item;
}
```

Al igual que ocurría con las lista de enteros, una variable de tipo `STRING_LIST` es un puntero a una estructura `ITEM_STRING_LIST` que representa el primer elemento de la lista de cadenas de carácteres. Estas estructuras tienen dos campos, uno de los cuales, que

[13] Puesto que los valores de tipo `INT_LIST` son punteros a valores de tipo `struct ITEM_INT_LIST`, al valor recibido como argumento por ésta y otras funciones de manejo de listas es realmente un puntero a un puntero a una estructura `ITEM_INT_LIST`. El doble nivel de indirección es requerido puesto que se va a modificar el valor del puntero incicial de la lista.

[14] El argumento que se pasa a esta función es un puntero a una variable de tipo `INT_LIST`.

recibe por nombre `data_item`, almacena un puntero a una cadena de caracteres, mientras que el otro, llamado `next_item`, almacena un puntero al siguiente elemento de la lista.

El conjunto de funciones que se han definido para tratar con este tipo de listas es equivalente al que se definió para trabajar con las listas de enteros. Así, se dispone de las siguientes funciones:

- `RESET_STRING_LIST`, que elimina la lista de cadenas de carácteres, teniendo cuidado de liberar toda la memoria ocupada, no sólo por las estructuras que componen cada elemnto, sino por las propias cadenas de caracteres a las que apunta cada una de dichas estructuras.

- `IS_EMPTY_STRING_LIST` comprueba si una lista de cadenas de carácteres esta vacía.

- `CONS_STRING_LIST`. Añade un elemento al principio de lista. Toma como parámetros un puntero a `INT_LIST` y un puntero a `char`. Esta función crea una nueva estructura `ITEM_STRING_LIST` y la enlaza al principio de la lista. Para rellenar el campo `data_item` se crea una nueva cadena de caracteres en la que se copia el contenido de la que se pasa como argumento, por lo que dicho argumento permanece sin cambios.

- `NEXTL_STRING_LIST`, elimina el primer elemento de una lista de cadenas de carácteres.

- `CAR_STRING_LIST`, devuelve la cadena de caracteres almacenada en el primer elemento de la lista de cadenas de carácteres.

6.3.3 Funciones de actualización del componente léxico

Para poder realizar una actualización consistente de la información del componente léxico, lo cual implica mantener la coordinación entre todas las lista que componen la estructura mediante la cual se implementa la variable `token`, se ha definido un conjunto de funciones que evitan la realización de actualizaciones *manuales* de tal variable.

Estas funciones han sido testeadas y se han mostrado seguras en el tratamiento de la información del componente léxico. Siempre que se deba modificar cualquier dato almacenado en `token` se debe hacer uso de ellas, evitando el acceso *ad hoc*, fuente potencial de problemas de inconsistencia de la información.

A continuación se muestra una lista de las funciones que se han considerado. Todas ellas reciben al menos un argumento: un puntero a una estructura `ice_lex_object`. Si hay argumentos adicionales, se indican en la descripción que se hace de cada una de ellas.

En primer lugar se van a mostrar las funciones que realizan el proceso de inicialición de la variable `token` y del conjunto de las listas de enteros y cadenas de carácteres:

- `reset_token`. Esta función vacía el contenido de una variable de tipo `ice_lex_token`, lo cual significa eliminar cada una de las listas que forman los distintos campos, para lo cual se utilizan las funciones `RESET_INT_LIST` y `RESET_STRING_LIST`.

- `new_column_token`. Función que crea un nuevo elemento al comienzo de cada una de las listas almacenadas en una variable de tipo `ice_lex_token`, estableciendo su valor a No-Aplicable. Esto se corresponde con la inicialización de un nuevo tipo de

componente léxico para una palabra[15]. Nótese la diferencia con la función precedente, que realizaba la incialización al comienzo del reconocimiento de una palabra.

- `copy_column_token`. Esta función actúa como la precedente, pero en vez de rellenar el valor de los nuevos elemntos de las listas a No-Aplicable, copia en ellos el contenido del elemento siguiente en la lista[16].
- `rm_column_token`. Elimina el primer elemento de todas las listas. Esto equivale a eliminar el último componente léxico detectado en el reconocimiento de una palabra determinada.

A continuación se muestran las funciones encargadas de la actualización de los valores almacenados en las listas de los diferentes campos incluidos en la variable `token`. Cada función recibe al menos como argumento un puntero a una estructura `ice_lex_token`.

- `set_wrd_token`. Recibe como argumento adicional un puntero a `char`, es decir, una cadena de caracteres. Copia dicha cadena en el campo `word` de la estructura `ice_lex_token`. Esto significa que la cadena pasada como argumento no sufre ningún tipo de modificación.
- `set_cat_token`. Recibe como argumento adicional un valor entero que indica la categoría la que pertenece la palabra. Esta función establece el primer elemento de la lista del campo `category` a dicho valor. Para ello utiliza las funciones `NEXTL_INT_LIST` y `CONS_INT_LIST`, con lo cual se machaca el valor No-Aplicable introducido en la inicialización del componente léxico llevada a cabo por la función `new_column_token`.
- `set_gen_token`. Recibe como argumento un valor entero que indica el género de la palabra cuando se reconoce como la categoría actual. El proceso es idéntico al que se realiza en la función anterior, salvo que la información que se actualiza es la de la lista almacenada en el campo `gender`.
- `set_num_token`. Esta función es como la precedente, excepto que el argumento que recibe es un entero que indica el número, por lo cual la información que se actualiza es la correspondiente a la de la lista almacenada en el campo `number`.
- `set_per_token`. Igual que la anterior pero referida a la persona verbal (campo `person`).
- `set_vtn_token`. Actualiza el valor del tiempo verbal, modificando el primer elemento de la lista del campo `verbal_tense`.

Una función adicional que ha sido preciso definir es `concat`. Su misión es la de realizar la concatenación de dos cadenas de caracteres que se pasan como argumentos. A diferencia de la función estándar `strcat`, la cual concatena físicamente la segunda cadena al final de la primera, esta función crea una nueva cadena de caracteres, reservando el espacio de memoria necesario, en el que copia secuencialmente los caracteres de ambas cadenas. El valor retornado es un puntero al primer carácter de esta nueva cadena de caracteres.

[15] Al tratarse de analizadores léxicos no deterministas, una misma palabra puede corresponderse con varios tipos de componente léxico distintos.

[16] Es decir, el que ocupaba anteriormente la cabeza de la lista.

6.3.4 Proceso de actualización de la información del componente léxico

Puesto que la variable `token` ha sido añadida por el programador al analizador generado por Flex, su valor debe ser inicializado y mantenido por el usuario. El esquema que se ha seguido ha sido el de ir actualizando el contenido de `token` tan pronto como esté disponible algún tipo de información relevante.

El valor retornado por la función `yylex` ha perdido gran parte de su significado, puesto que ahora tan sólo se consideran dos posibles opciones:

- Que el valor devuelto sea 0, en cuyo caso indica que se ha alcanzado el final del fichero sobre el cual se está realizando el reconocimiento de los componentes léxicos.
- Cualquier otro valor indica el reconocimiento de alguna palabra. Por defecto el valor devuelto se establece a 1, aunque su valor concreto es irrelevante, basta con que sea distinto de 0. El reconocimiento de una palabra puede tener tres orígenes diferentes:
  - La detección de una correspondencia de la palabra con alguno de los tipos de componentes léxicos declarados en el analizador sintáctico.
  - El reconocimiento como una palabra errónea, esto es, una palabra que está mal construida según las reglas léxicas[17].
  - Una palabra desconocida.

El valor contenido en la variable `yylval` es irrelevante, ya que toda la información semántica concerniente al proceso de reconocimiento de componentes léxicos se encuentra almacenada en la variable `token`.

6.3.5 Actualización mediante reglas léxicas

Según se van aplicando las reglas léxicas, se va obteniendo información acerca de la palabra que está siendo reconocida.

El reconocimiento de los lexemas

La detección del lexema de una palabra indica que potencialmente se ha encontrado un nuevo componente léxico en que encuadrarla. Por ello, las reglas encargadas de realizar el reconocimiento de los lexemas llaman a la función de inicialización `new_column_token`.

En estas reglas, ya se está en condiciones de poder establecer la siguiente información concerniente al nuevo análisis que se está realizando de la palabra:

- La categoría de la palabra.
- El lema. La información del lema puede establecerse ya que una vez que se conoce la categoría y el lexema, se pueden aplicar las reglas léxicas correspondientes para construir el lema.

Para establecer esta información en la variable `token` se utilizan sendas llamadas a las funciones `set_cat_token` y `set_lem_token`.

[17] Esto último se debe generalmente a errores tipográficos o del reconocedor óptico de caracteres en caso de que haya sido escaneada.

El reconocimiento de los sufijos

En cada una de las condiciones de arranque a las que se envía al reconocedor, se obtiene algún tipo de información relevante que debe ser almacenada en la variable `token`. Por ejemplo, cuando se alcanza una de las condiciones que reconocen el género de una palabra, se obtiene la información que determina si el género es el masculino o el femenino. Lo mismo ocurre cuando se alcanza una condición referente al número con respecto al singular o al plural. En el caso de los verbos, las condiciones de arranque que identifican el tiempo verbal y la persona.

Por tanto, cuando se alcanza una condición de arranque, se debe realizar una llamada a la función que actualiza la lista en la cual se almacena la información que se acaba de obtener.

El encadenamiento de las condiciones de arranque conlleva la actualización sucesiva de diferentes listas.

Aquella información que no se obtiene en el proceso de reconocimiento de una palabra queda establecida a No-Aplicable, puesto que así se estableció cuando fue inicializada mediante las llamadas a `new_column_token` o `copy_column_token`. Por ejemplo, los sustantivos no tienen ni tiempo ni persona verbal, por lo que los campos `person` y `verbal_tense` tendrán valor `NA`.

6.3.6 Actualización no determinista

La actualización no determinista de los componentes léxicos está relacionada con la utilización de las condiciones de arranque correspondientes a la salida y a la detección de errores.

La condición de salida

A esta condición de arranque se llega a partir de la regla sin condición de arranque cuyo patrón es el punto. Con ello se indica que ya se han reconocido todas los posibles componentes léxicos para una palabra dada.

En este momento se dispone de la información concerniente a la porción de texto que constituye la palabra, por lo que se puede establecer el campo `word` de la variable `token`. Ciertamente, esta información ya se conoce cuando se finaliza la cadena de condiciones de arranque para cada análisis léxico de la palabra. Sin embargo, este es el único lugar en el que se puede determinar con seguridad que no se va a realizar ningún análisis más, a nivel léxico, de esta palabra. Por tanto es el lugar idóneo para guardar esta información, ya que se garantiza que sólo será actualizada una vez para cada palabra. Pero lo que es aún más importante, es la única condición de arranque por la que se garantiza que pasará el análisis de toda palabra, por lo que las dos aternativas serían:

- Actualizar el campo `word` en cada una de las condiciones de arranque que finalizan una cadena de reconocimiento de reglas léxicas.
- Actualizarlo en la condición de salida, por la que siempre se pasa.

Como se ve, la solución más simple y eficiente consiste en realizar el almacenamiento de la información concerniente a la cadena de caracteres que conforma la palabra que acaba de ser reconocida en la condición <S> de salida.

Si se llega a esta condición con las listas de enteros y de cadenas de carácteres de los campos de la variable `token` vacías, significa que la palabra no ha sido reconocida como una de las incorporadas al analizador léxico. En este caso, se utiliza la categoría Desconocido para la palabra. Para establecerla, es necesario llamar a la función `new_column_token` y posteriormente a `set_cat_token`, pasándole a esta última, además de la dirección de `token`, el valor `UKN` que identifica a la categoría Desconocido.

Cuando se alcanza la condición de salida, el analizador léxico devuelve el control al sintáctico. En este momento, este último ya dispone en la variables `token` de toda la información concerniente a la palabra que se ha podido obtener.

La condición de error

Esta condición se alcanza cuando una palabra ha comenzado a ser reconocida aplicando algunas reglas léxicas, pero se ha detectado que no cumple alguna de ellas. En tal caso, en las acciones correspondientes a la regla incluida en esta condición de arranque se llama a la función `rm_column_token` para eliminar la información del análisis erróneo de la variable `token`.

En caso de que se desee realizar una depuración del analizador léxico, podría ser interesante mantener tal información, pero sumando al campo categoría un valor `ERR`[18] que indique que dicha información es errónea.

6.3.7 Recuperación de información del componente léxico

Una vez que ha sido realizado el análisis léxico de una palabra, el analizador sintáctico generado por ICE debe ser capaz de acceder a al información almacenada en la variable `token`. Para ello es necesario definir una serie de funciones de acceso a los distintos valores y también realizar algunos cambios en el generador de analizadores de ICE.

Funciones de recuperación de información

Puesto que los analizadores generados por ICE están escritos en código LE-LISP,mientras que el analizador léxico está escrito en C, es necesasrio escribir un conjunto de funciones que permitan acceder a los valores de una estructura C de tipo `ice_lex_token` desde funciones LE-LISP. Puesto que la manipulación de los campos de una estructura C desde un programa LE-LISP se presenta ciertamente complicada, se ha decidido utilizar una función de acceso para cada uno de los campos de la estructura. Todas estas funciones toman como argumento un puntero a un componente léxico:

- `get_category`. Esta función obtiene el valor del primer elemento de la lista de categorías almacenada en el campo `category` de la variable `token`.
- `get_gender`. Obtiene el valor del primer elemento de la lista donde se almacena el género de la palabra.
- `get_number`. Obtiene el primer valor de la lista del campo `number` de la palabra.

[18] Este valor **ERR** deberá ser igual a 1 más el mayor valor numérico utilizado para designar una categoría. Con ello, el analizador sintáctico, o el programa especial de depuración del analizador léxico, puede determinar que el campo **category** no contiene un valor válido. Restando **ERR** de su valor se obtendrá la categoría fallida que se intentó reconocer.

- `get_person`. Permite obtener el valor almacenado en el primer elemento de la lista del campo `person`.
- `get_verbal_tense`. Se utiliza para obtener el primer tiempo verbal almacenado en la lista del campo `verba_tense`.
- `nextl_token`. Esta función elimina el primer elemento de las diferentes listas de enteros y cadenas de carácteres, realizando las oportunas llamadas a las funciones `NEXTL_INT_LIST` y `NEXTL_STRING_LIST`. Su utilización permite acceder a los restantes elementos de las listas mediante la utilización de las funciones `get_`*xxx*.

Es necesario definir una función que permita obtner la dirección de la variable `token`. Puesto que esta varible sólo debe ser accedida cuando ha finalizado el análisis léxico de una palabra, o lo que es lo mismo, cuando termina la ejecución de la función `yylex`, se ha optado por definir una nueva función denominada `yygalenalex` que se encargará de:

- Realizar la llamada a `yylex()` para que tenga lugar el análisis léxico de una palabra.
- Devolver la dirección de la variable `token` cuando el valor devuelto por `yylex` sea distinto de 0.
- Devolver el valor `NULL` cuando la función `yylex` devuelva como resultado el valor 0 (fin de fichero).

El fichero `output.c`

En el fichero `output.c` se genera, entre otras cosas, el código que será localizado en el fichero de salida `yacc`*lang*`.tab.c`. Concretamente, en la función `Le_Lisp_ouput_token_translations` se genera la macro `{ice}:translate` que realiza la interpretación del valor devuelto por el analizador léxico a través de `yylex`. En nuestro caso, en vez de llamar a `yylex` llamaremos a `yygalenalex`. Por ahora, para realizar el análisis sintáctico, al analizador le es suficiente con conocer la categoría a la cual pertenece la palabra. En fases más avanzadas, la incorporación del análisis semántico hará necesario el acceso al resto de la información almacenada en la variable `token`.

Por consiguiente, la definición de la función `Le_Lispoutput_token_translation` queda como se muestra en el siguiente fragmento de código:

```
void
Le_Lisp_output_token_translations()
{
  register int i, j;
/*   register short *sp; JF unused */
  if (translations)
    {
      fprintf(ftable,"\n(defextern _get_category (t) fix)");
      fprintf(ftable,
              "\n(dmd {ice}:translate (x)\n
                      `(let ((x (_get_category ,x)))\n
                            (if (le x %d) (vref {ice}:yytranslate x) %d)))\n",
              max_user_token_number, nsyms);
      if (ntokens < 127)  /* play it very safe; check maximum element value.  */
```

```
        fprintf(ftable, "\n(defvar {ice}:yytranslate #[     0");
      else
        fprintf(ftable, "\n(defvar {ice}:yytranslate #[     0");
      j = 10;
      for (i = 1; i <= max_user_token_number; i++)
        {
          putc(' ', ftable);
          if (j >= 10)
            {
              putc('\n', ftable);
              j = 1;
            }
          else
            {
              j++;
            }
          fprintf(ftable, "%6d", token_translations[i]);
        }
      fprintf(ftable, "\n])\n");
    }
  else
    {
      fprintf(ftable, "\n(dmd {ice}:translate (x) `(,x))\n");
    }
}
```

El fichero `ice.ll`

Este fichero pertenece a la parte del intérprete de ICE. Se puede decir sin lugar a dudas que su código constituye el núcleo fundamental del intérprete. Entre sus funciones se encuentra la de cargar el código del analizador léxico y realizar las oportunas llamadas a la función `yylex`. En esta implementación, se sustituirán tales llamadas por invocaciones a la función `yygalenalex`, definida anteriormente.

Para ello, en la función `{ice}:load-scanner`, inmediatamente depués de haber cargado el analizador léxico, se debe declarar como externa la función `yygalenalex`, junto con el resto de las funciones utilizadas en la interacción con el analizador léxico. Esta declaración se realiza mediante `defextern`.

Las funciones en las que hay que cambiar las llamadas a `_yylex`[19] por llamadas a `_yygalenalex` son las siguientes:

- `{ice}:loop-from`, función encargada de llamar a la función de reconocimiento de componentes léxicos hasta que se termina el fichero.
- `{ice}:begin`, función que inicia el proceso de creación de los items e itemsets.
- `{ice}:run`, función encargada de lanzar el proceso de análisis sintáctico sobre un fichero determinado.
- `{ice}:insert-change`, función encargada de insertar en el bosque compartido una modificación incremental del análisis.

[19] El nombre de la función externa a LE-LISP que se correponde con la función C del analizador léxico generado por Flex.

6.4 Integración con ICEeditor

A continuación se va a mostrar cómo se realiza el enlace entre ICEeditor y los analizadores sintácticos generados por ICE. Se van a distinguir dos casos básicos: uno concerniente a la realización de un análisis total del texto que se encuentra en el editor de ICEeditor, y otro que involucra la realización de un análisis incremental tomando como base las modificaciones llevadas a cabo sobre dicho texto.

Antes de comenzar cualquier tipo de análisis, se hace preciso inicializar el intérprete ICE. Para ello se realiza una llamada a la función `init-ice`, la función de inicialización del intérprete, desde la función `ICEeditor`, que es la encargada de proceder con la inicialización de ICEeditor.

6.4.1 Análisis total

El proceso para la realización de un análisis sintáctico completo del texto ubicado en el editor de ICEeditor consta de los dos pasos siguientes:

1. Cargar el lenguaje que se está utilizando. Para ello se llama a la función {`ice`}:`load`, que toma como argumento una cadena de caracteres que contiene el nombre del lenguaje. Una llamada a ({`ice`}:`load` "*lang*") provoca que el intérprete ICE busque, en el directorio indicado en la función {`ice`}:`user-language` en la cual se declaró el lenguaje *lang*, los ficheros `yacc`*lang*`.tab.c` y `lex`*lang*. Esta operación se realiza únicamente la primera vez que se llama al parser de un lenguaje que no había sido cargado previamente.

2. Llamar a la función encargada de realizar el análisis sintáctico del texto, en base a las tablas y funciones contenidas en el fichero `yacc`*lang*`.tab.c`. Esta función se denomina {`ice`}:`run` y toma como argumento una cadena de caracteres con el nombre del fichero a analizar. La extesión de dicho fichero deberá ser *.lang*. Como segundo argumento opcional puede tomar el átomo `'trace`, en cuyo caso el usuario obtiene una traza de los diferentes estados por los que va pasando el analizador.

Cuando el usuario solicita que el texto en ICEeditor sea reanalizado completamente, se activa la función {`ICEeditor`}:`parser-all`, la cual realiza las llamadas a las funciones mencionadas anteriormente. Como la función {`ice`}:`run` toma el nombre de un fichero en disco, y el texto en el editor no tiene por qué estar actualizado en su correspondiente fichero en el momento de realizar el análisis, el método `parser-all` realiza una copia de todo el texto contenido en el editor en un fichero localizado en el directorio `/tmp`. Es sobre este fichero sobre el que se lleva a cabo el análisis sintáctico. Todo este trabajo con ficheros temporales es transparente para el usuario.

6.4.2 Análisis incremental

Para realizar un análisis incremental, es preciso indicarle a ICE los componentes léxicos que han sufrido modificaciones respecto al último análisis. La información que ICE precisa conocer es la siguiente:

- El itemset que ha sufrido la modificación. Esto es, el componente léxico cuyo texto se ha modificado[20].
- El nuevo texto asociado a dicho itemset.
- El siguiente itemset[21].

Para poder proporcionar tal información, ICEeditor transforma todas las operaciones de edición de componentes léxicos en operaciones de modificación, las cuales se almacenan en el campo `operations` de la estructura {`ICEeditor`}, tal como se indica en el capítulo 2.

Cuando el usuario solicita la realización de un análisis incremental del texto, se activa el método `parser` asociado a la estructura {`ICEeditor`}. Este método realiza las siguientes acciones:

1. Llama a la función {`ice`}:`old` para obtener el primer itemset resultado del análisi anterior.
2. Para cada componente léxico que ha sido modificado:
   (a) Recupera su itemset realizando sucesivas llamadas a la función {`ice`}:`get-next`.
   (b) Busca el itemset siguiente a la secuencia contigua de componentes léxicos modificados.
   (c) Crea un nuevo elemento en la lista que almacena los cambios sufridos por los componentes léxicos desde el último análisis. Para ello se realizan llamadas a las funciones {`ice`}:`changes`[22] y {`change`}:`new`[23], como se muestra en el siguiente fragmento de código[24]:

$$(\texttt{newr}\ \{\texttt{ice}\} : \texttt{changes}\ (\texttt{change} : \texttt{new}\ \ \begin{array}{l}\text{"}new\_text\text{"}\\ itemset\\ next\_itemset))\end{array}$$

   en donde *"new_text"* representa el nuevo texto del componente léxico después de ser modificado, *itemset* indica el itemset que ha sido modificado y *next_itemset* el siguiente itemset a la cadena contigua de comonentes léxicos modificados..
3. Realiza una llamada a {`ice`}:`incremental`, la función que realiza el análisis incremental. Esta función ha sido modificada para pasar al analizador léxico la posición de inicio de cada modificación.

[20] Cada componente léxico tiene asociado un único itemset.
[21] Este dato tiene sentido si se borran varios componentes léxicos consecutivos.
[22] Esta función es la encargada de añadir un nuevo elemento a dicha lista.
[23] Esta función crea un nuevo componente de la lista de cambios.
[24] La función **`newr`** concatena físicamente a una lista un nuevo elemento.

Capítulo 7

Guía para el usuario

En este capítulo se va a mostrar la aplicación ICEeditor desde el punto de vista del usuario. Se van a describir los elementos gráficos que la componen y cómo puede interactuar con ellos el usuario.

7.1 Iniciando ICEeditor

Existen al menos dos formas de comenzar a trabajar con ICEeditor, ambas equivalentes, puesto que el entorno de trabajo que se obtendrá será el mismo.

La primera forma consiste en lanzar primero AÏDA o MASAÏ y una vez que aparece la patalla principal de control, cuyo aspecto se muestra en la figura 7.1[1], seguir alguna de las siguientes opciones:

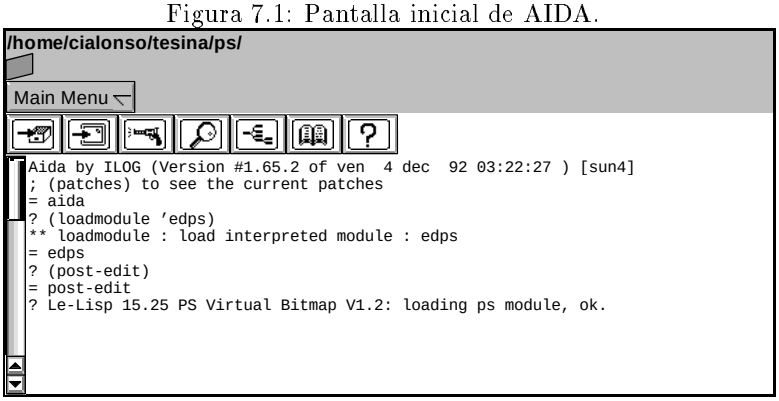
1. Llamar a la función `init-ICEeditor`, que se encargará de la inicialización de ICE y de ICEeditor, así como de la activación de este último.
2. Inicializar primero ICE, mediante una llamada a la función `init-ice` y posteriormente inicializar la aplicación ICEeditor siguiendo alguno de los métodos que se muestran a continuación:
   (a) Llamar a la función `load` de LE-LISP para cargar el programa `ICEeditor.ll`. El argumento de `load` deberá ser una cadena de caracteres con la ruta de dicho fichero. Por ejemplo:

   ```
   (load "tesina/code/aida/code/ICEeditor.ll")
   ```

   (b) Utilizar la combinación de teclas <CONTROL>L seguida de la ruta del fichero. En este caso no es necesario poner la extensión `.ll`. Por ejemplo:

[1] Para capturar las imágenes de las aplicaciones AÏDA se ha utilizado el módulo **`edps`** [ILOG 91c], que no se incluye de modo estándar en el entorno incial. Para cargarlo es necesario teclear **`(loadmodule 'edps)`**. Una vez cargado, este módulo proporciona acceso a la función **`post-edit`**. Una llamada a esta función, que no tiene argumentos, lanza una aplicación AÏDA que permite almacenar cualquier imagen AÏDA en un fichero PostScript. La imagen almacenada no es un bitmap, sino que se transforman todos los elementos gráficos utilizados para construir la imagen en sus equivalentes PostScript. Con ello se consigue, por ejemplo, que al imprimirlo en una impresora láser, se utilicen las fuentes PostScript para el texto, con lo que se dispondrá de toda la resolución de la impresora, independientemente de la resolución de la pantalla en la cual fue capturada la imagen. Sin embargo, se pueden plantear problemas a la hora de realizar la captura exacta de la imagen que se desea pasar a PostScript. Concretamente, para capturar las imágenes que aparecen en este trabajo, se ha utilizado una versión de AÏDA ejecutándose en una workstation Sun SPARC 10, pero Utilizando como servidor X la consola de un IBM RS/6000 mod. 550 con el gestor de ventanas Motif en la mayoría de los casos, excepto para la captura de los iconos, en que se hizo preciso sustituir **`mwm`** por el gestor de ventanas **`twm`**.

Figura 7.1: Pantalla inicial de AÏDA.




```
^Ltesina/code/aida/code/ICEeditor
```

(c) Llamar al editor de ficheros de AÏDA para editar el programa `ICEeditor.ll` y utilizar la opción *eval* del submenú *actions* para evaluar el contenido de dicho fichero. Una vez inicializada, se puede activar la aplicación ICEeditor mediante una llamada a la función `ICEeditor`.

La segunda forma de lanzar ICEeditor consiste en utilizar una versión previamente compilada de la aplicación. Para activarla, basta teclear el nombre del ejecutable obtenido[2]. Dicho ejecutable lanzará AÏDA e inmediatamente ICEeditor. Este método presenta las siguientes ventajas:

- Una mayor simplicidad al eliminar la necesidad de introducir expresiones LE-LISP.
- No es necesario tener una versión de AÏDA para poder ejecutar ICEeditor[3].

Una vez que ha sido activado ICEeditor, aparecerá en pantalla la ventana de espera que se muestra en la figura 7.3, que avisa al usuario de que se está realizando la carga del módulo `textedit`. Como esta operación puede llevar cierto tiempo[4], se advierte de ello en esta ventana para que el usuario no se impaciente.

7.2 Uso general de ICEeditor

Una vez que se ha cargado el módulo `textedit`, aparecerá la ventana principal de ICEeditor (figura 7.4). Esta ventana está formada por los siguientes elementos gráficos:

- Un editor de textos que ocupa la parte central de la ventana.
- Un scroller, con barras de desplazamianto horizontal y vertical, que enmarca al editor de textos.
- Un menú situado en la parte superior del editor.

[2] Dicho ejecutable debe estar accesible en el path del usuario. El fichero debe tener permiso de ejecución.

[3] El ejecutable incluye todos los recursos de AÏDA necesarios para poder ejecutar ICEeditor, pero no incluye el entorno de desarrollo.

[4] Típicamente del orden de una docena de segundos.

Figura 7.2: Editor de ficheros de AÏDA.

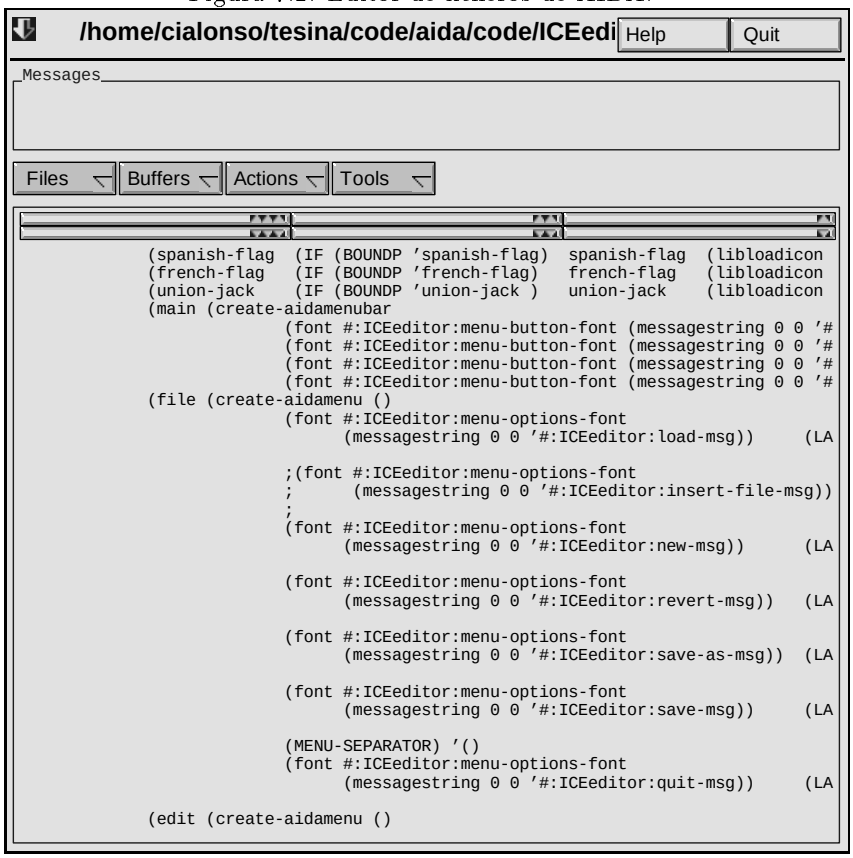
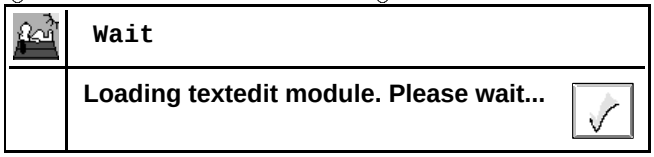
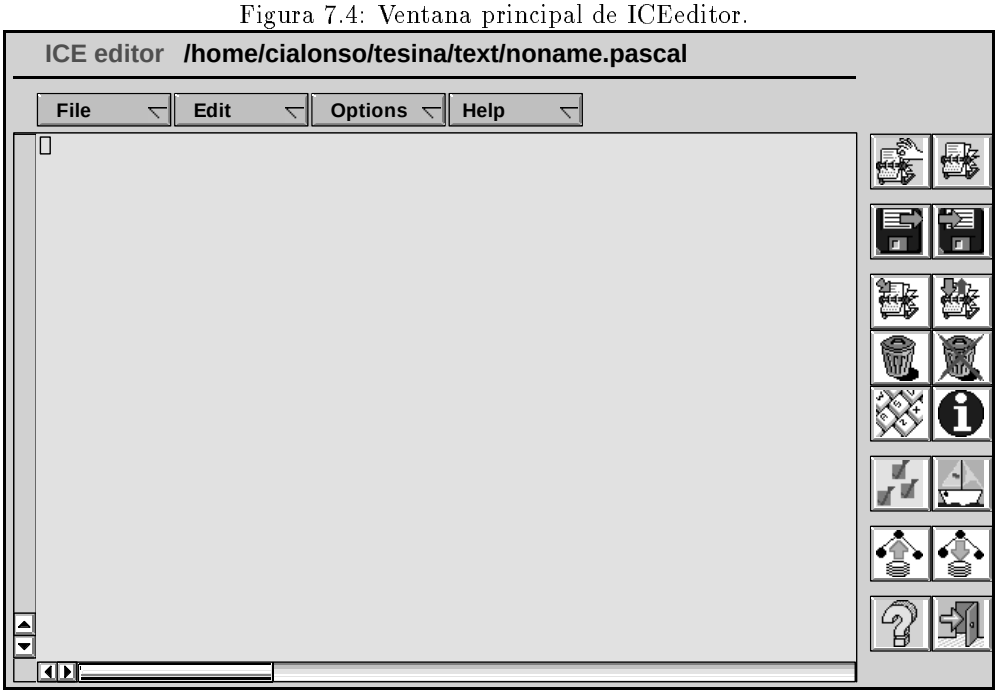


Figura 7.3: Aviso inicial de la carga del módulo `textedit`.



- Una barra de botones ubicada en el borde derecho de la ventana.
- Un pequeño editor en la parte superior de la ventana que contiene el nombre del fichero que se está utilizando actualmente en ICEeditor.

Figura 7.4: Ventana principal de ICEeditor.



El usuario puede comenzar a escribir directamente en el editor el texto que posteriormente va a ser analizado y, en su caso, editado para un posterior procesamiento incremental.

Además de la ventana principal, durante la sesión de trabajo con ICEeditor aparecerán otras dos ventanas, denominadas *ICE messages* y *ICE navigate*, respecitvamente. La primera se utiliza para mostrar los mensajes generados durante el proceso de análisis, mientras que la segunda permite que el usuario naveque por la estructura del bosque compartido generada tras el análisis. En la figura 7.5 se muestra un aspecto de la aplicación ICEeditor con las tres ventanas activas.

7.2.1 Carga de un fichero

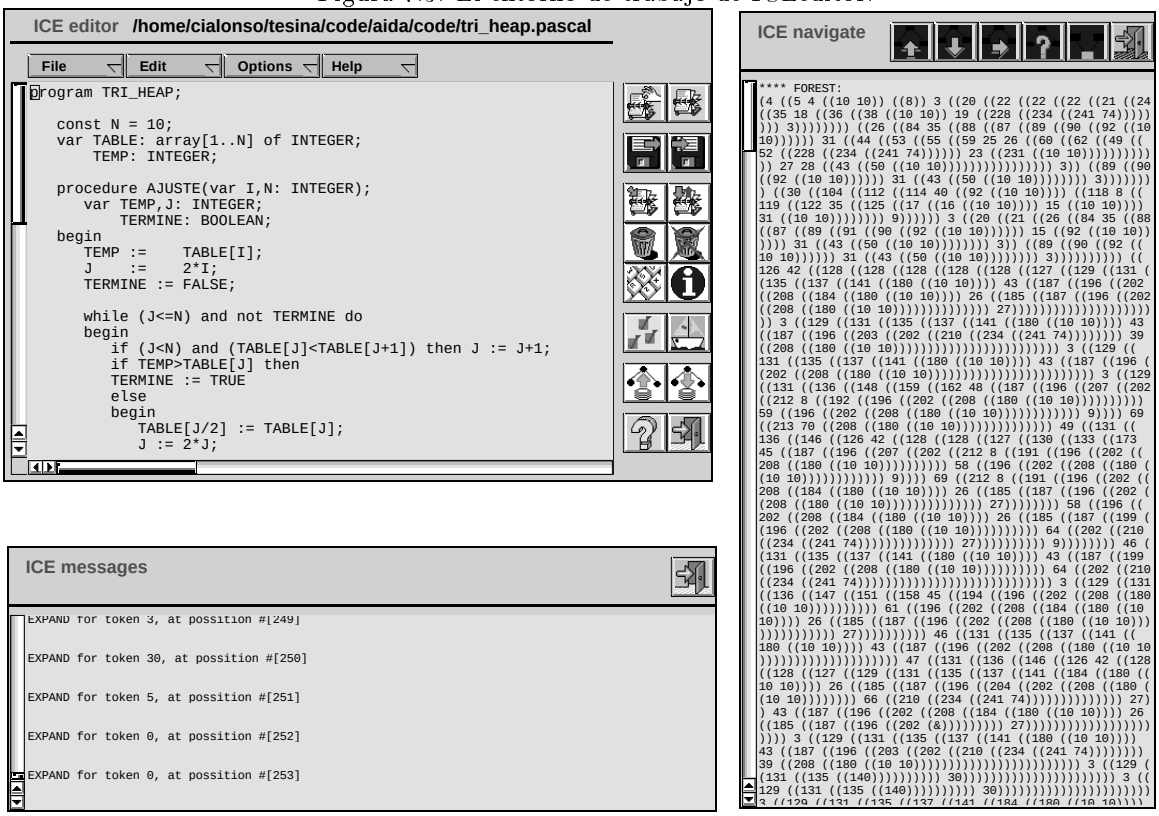
Se puede cargar un fichero de texto previamente tecleado[5] utilizando bien el botón de *load*[6] o bien la opción *load* del menú *file*. Una vez que se ha seleccionado la opción load por cualquiera de los dos procediminetos, aparecerá una ventana de diálogo para preguntar el nombre del fichero que se desea cargar. Dicha ventana se muestra en la figura 7.11.

Se puede editar directamente el nombre del fichero que aparece en la parte superior de la ventana. Este nombre será ofrecido como opción por defecto en todas aquellas

[5] Utilizando ICEeditor o cualquier editor de texto como puede ser Emacs o vi.

[6] Como se verá posteriormente, ICEeditor soporta múltiples idiomas para interactuar con el usuario. Sin embargo, cuando en este trabajo se nombre algún elemento gráfico, como por ejemplo una opción de menú, se utilizará el nombre que recibe dicho elemento en la versión en inglés de ICEeditor.

Figura 7.5: El entorno de trabajo de ICEeditor.



operaciones que involucren el manejo de nombres de ficheros.

7.2.2 El editor de textos

El contenido del fichero cargado en el editor puede ser editado libremente por el usuario. El editor incorporado en ICEeditor posee características avanzadas de edición, del tipo de las presentes en el editor Emacs.

Se puede utilizar el ratón para señalar la posición en la que se desea que se situe el cursor. Es posible realizar operaciones de *copiar y pegar*[7] utilizando el ratón. Para ello se han de realizar los siguientes pasos:

1. Pulsar con el botón izquierdo del ratón en el primer carácter que se desee copiar.

2. Sin soltar el botón izquierdo, arrastrar el ratón hasta que quede resaltado toda la porción de texto que se desea editar. Entonces se debe soltar el botón. El texto pasará al buffer interno del editor y dejará de estar resaltado.

3. Situar el ratón en la posición a la que se desea copiar el texto seleccionado y pulsar el botón central del ratón.

Como ya se ha dicho anteriormente, se pueden utilizar combinaciones de teclas del estilo de las disponibles en Emacs para realizar operaciones con bloques y desplazamientos a saltos a lo largo del texto. Pulsando las teclas <ESC>? se obtiene una ventana de ayuda como la que se muestra en la figura 7.6.

En la tabla 7.1 se reunen las teclas que se pueden utilizar en combinación con la tecla CONTROL.

En la tabla 7.2 se muestan las combinaciones de teclas basadas en la pulsación de la tecla ESC.

7.2.3 Grabación de un fichero

Cuando se ha terminado de editar un fichero,o en cualquier otro momento que se considere oportuno, se puede guardar en disco. Para ello se debe seleccionar la opción *save* del menú *file* o bien presionar el botón *save* con el botón izquierdo del ratón.

Al seleccionar la opción de salvar, si no se le ha dado nombre al fichero, aparecerá una ventana de diálogo en la que se introducirá dicho nombre.

Una vez que se ha dado nombre a un fichero, para cambiarlo se puede utilizar la línea situada sobre el editor, o bien seleccionar la opción *save as* del menú *file*, tras lo cual aparecerá en pantalla una ventana de diálogo similar a la que aparece cuando se procede a cargar un fichero en ICEeditor. En esta ventana se nos ofrece el nombre que aparece en la parte superior del editor como opción por defecto para nombrar al fichero. Si se desea se puede cambiar dicho nombre, así como seleccionar el directorio en el cual se desea que sea guardado el fichero.

[7] *Copy and paste.*

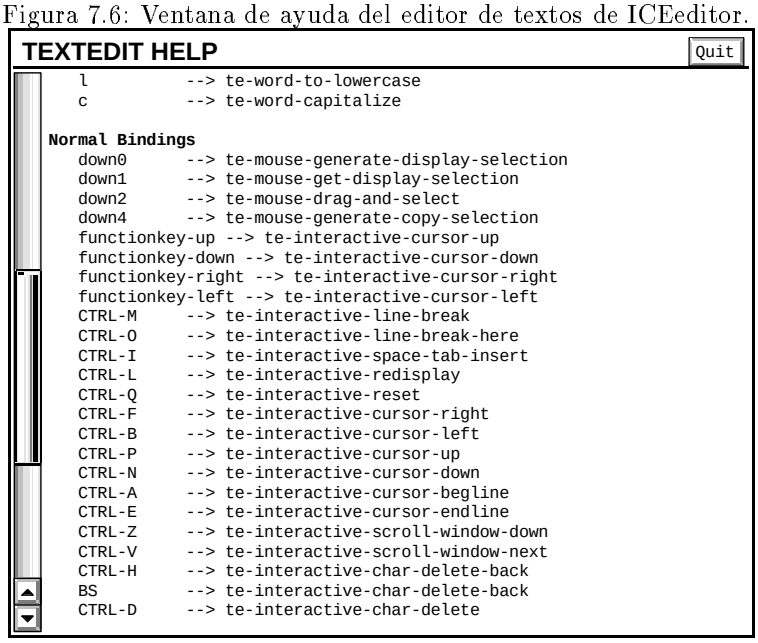
Tabla 7.1: Combinaciones con la tecla <CONTROL> en ICEeditor.

<CONTROL>SPACE	Establece una marca en el texto. Dicha marca indicará un punto de referencia al realizar operaciones de bloques.
<CONTROL>W	Borra los caracteres comprendidos entre la última marca establecida y la posición en la que se pulsó esta combinación de teclas. El texto borrado pasa al buffer y puede ser recuperado posteriormente. Sólo se puede recuperar el último bloque borrado.
<CONTROL>Y	Recupera el texto almacenado en el buffer, situándolo en la posición en que se pulsaron estas teclas.
<CONTROL>H	Borra un carácter.
<CONTROL>M	Introduce una ruptura de línea en el texto, desplazando el cursor al comienzo de la línea siguiente.
<CONTROL>O	Introduce una carácter de ruptura de línea, pero no desplaza la posición del cursor.
<CONTROL>I	Inserta un tabulador en el texto.
<CONTROL>L	Redibuja el contenido del editor.
<CONTROL>Q	Borra el contenido del editor. Pide confirmación.
<CONTROL>F	Desplaza el cursor hacia la derecha.
<CONTROL>B	Desplaza el cursor hacia la izquierda.
<CONTROL>P	Desplaza el cursor hacia arriba.
<CONTROL>N	Desplaza el cursor hacia abajo.
<CONTROL>A	Desplaza el cursor al comienzo de la línea.
<CONTROL>E	Desplaza el cursor al final de la línea.
<CONTROL>Z	Desplaza el texto una línea hacia abajo.
<CONTROL>V	Desplaza el texto hacia abajo un número de líneas equivalente al tamaño vertical del editor.
<CONTROL>D	Borra el carácter situado inmediatamente a la derecha del cursor.

Tabla 7.2: Combinaciones con la tecla <ESC> en ICEeditor

<ESC>f	Mueve el cursor al comienzo de la siguiente palabra. Una palabra, desde el punto de vista del editor de textos incorporado en ICEeditor, es una secuencia arbitraria de caracteres delimitada por signos de puntuación: espacios, comas, puntos, ...
<ESC>b	Mueve el cursor al comienzo de la palabra anterior.
<ESC>d	Borra la parte de la palabra situado a la derecha del cursor.
<ESC><CONTROL>H	Borra la parte de la palabra situada a la izquierda del cursor.
<ESC><BACKSPACE>	Como la anterior.
<ESC>Z	Mueve el texto una línea hacia arriba, sin modificar la posición del cursor.
<ESC>V	Desplaza el texto hacia arriba un número de líneas equivalente a la altura del editor.
<ESC><	Desplaza la posición del cursor al principio del texto.
<ESC>>	Desplaza la posición del cursor al final del texto.
<ESC>?	Muestra una ventana de ayuda con las combinaciones de teclas válidas.
<ESC>u	Pasa a mayúsculas una palabra.
<ESC>l	Pasa a minúsculas una palabra.
<ESC>c	Pone en mayúsculas la primera letra de la palabra siguiente.

Figura 7.6: Ventana de ayuda del editor de textos de ICEeditor.



7.2.4 Análisis de un texto

Para analizar el texto del editor se puede utilizar bien la barra de botones o bien el submenú *parser* de la barra de menú.

Se dispone de dos opciones a la hora de realizar el análisis sintáctico del texto del editor:

- *Análisis total del texto.* Con esta opción se realiza un análisis completo del texto, desechando cualquier información que pudiera estar disponible como resultado de análisis anteriores. El lenguaje adecuado al fichero que se está editando se carga automáticamente, tomando como base la extensión del fichero.

- *Análisis incremental.* Esta opción permite realizar un análisis incremental del texto almacenado en el editor. Se utiliza la información del análisis anterior y la que proviene de las operaciones de edición de componentes léxicos realizadas por el usuario para guiar el nuevo proceso de análisis.

Durante la realización del análisis se borrará el texto de los componentes léxicos que habían sido marcado para borrar. Una vez terminado el análisis, ya no habrá texto resaltado, puesto que todas las operaciones de edición realizadas con anterioridad ya han sido incorporadas en el nuevo análisis sintáctico.

La realización de un análisis sintáctico incremental sobre un texto que no ha sufrido modificaciones desde el último análisis, no tendrá ningún efecto. Una petición de análisis total siempre reanaliza completamente el texto, incluso en el caso de que permanezca intacto desde el último análisis.

7.2.5 Edición de los componentes léxicos

Una vez que el texto ha sido analizado, el texto del editor se rige por la estructura de componentes léxicos que ha sido creada en el proceso de análisis. Es por ello que el texto ya no puede ser editado libremente sino que todas las operaciones de edición deben adaptarse a la estructura de componentes léxicos.

Existen tres operaciones auténticas de edición, que son:

- **Inserción**. Para insertar un nuevo componente léxico antes de un componente léxico existente.
- **Modificación**. Para modificar el texto de un componente léxico.
- **Borrado**. Para eliminar un componente léxico en el siguiente análisis.
- **desborrado**. Para retornar a su estado inicial un componente léxico previamente borrado.

Adicionalmente, se dispone de operaciones que actúan sobre el editor de componentes léxicos pero que no intervienen en la realización de los análisis incrementales. Estas operaciones son:

- **Información**. Permite obtener información de un componente léxico.
- **Sin acción**. Permite indicar que no se va a realizar ninguna operación de edición.

La realización de cualquiera de estas operaciones involucra un proceso que consta de las siguientes fases:

1. Activar la operación que se desea realizar. Para ello el usuario debe de realizar alguna de las acciones siguientes:
   - Pulsar con el botón izquierdo del ratón en el botón correspondiente a la operación que se desea realizar.
   - Abrir el menú *edit* y seleccionar la opción que se corresponde con la operación que se desea llevar a cabo.
2. En el caso de una modificación o borrado, indicar el componente léxico sobre el cual se desea realizar dicha operación. Lo mismo ocurre en el caso de que se dese obtener información acerca de un componente léxico. Cuando se trata de realizar una inserción, se indicará el componente léxico antes del cual se va a insertar el nuevo componente léxico[8]. ICEeditor facilita esta tarea permitiendo que el usuario pulse con el botón izquierdo del ratón en cualquier posición del texto. La aplicación es capaz de determinar automáticamente el componente léxico al cual pertenece el carácter. El texto de dicho componente léxico aparecerá resaltado en los colores correspondientes a la operación que se esté realizando. Estos colores pueden ser modificados por el usuario. Por defecto, ICEeditor arranca con los siguientes colores:
   - Fondo azul y caracteres negros para el texto comprendido en operaciones de inserción.

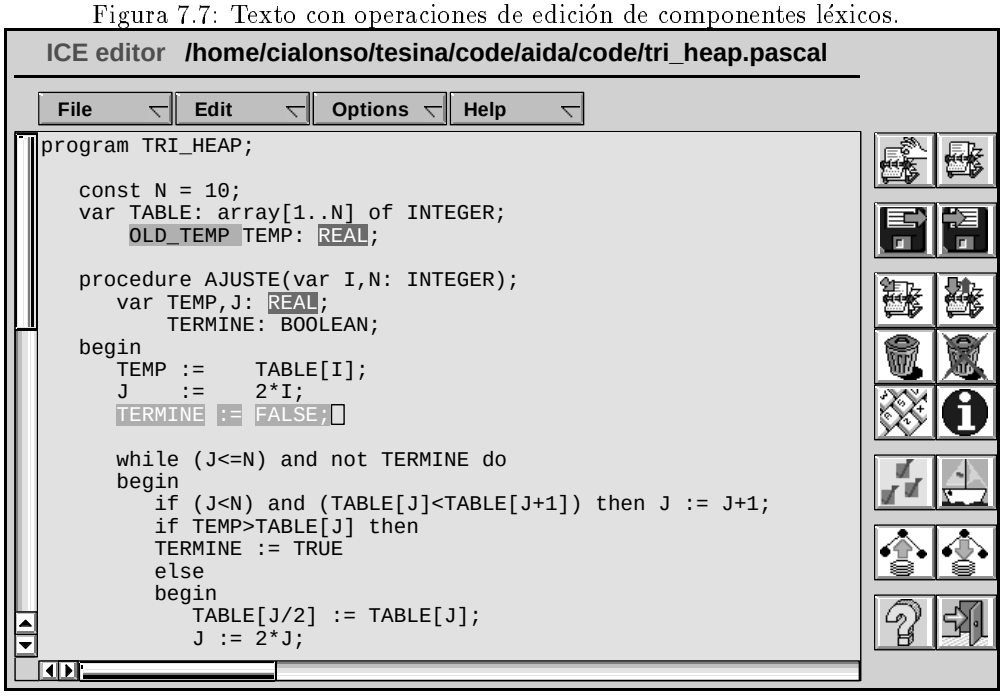
[8] Es decir, se indicará un componente léxico y el texto del nuevo componente léxico será insertado delante de dicho componente léxico.

- Fondo de color magenta y caracteres blancos para el texto involucrado en las operaciones de modificación.
- Fondo naranja y caracteres blancos para el texto de los componentes léxicos que han sido borrados.
- Fondo amarillo y caracteres en negro para el texto del componente léxico del cual se está solicitando información.

3. En el caso de las operaciones de inserción y modificación, el usuario puede introducir texto mediante el teclado. En la figura 7.7 se muestra a ICEeditor con un texto sobre el cual se han realizado operaciones de inserción, modificación y borrado. Cada una de estas operaciones presenta ciertas particularidades:

   - En el caso de una modificación, el cursor permanecerá situado en el carácter sobre el que se pulsó el ratón, con todo el texto del componente léxico resaltado. Se puede insertar y borrar texto libremente dentro de la zona resaltada.
   - En el caso de una inserción, el cursor se situará en el primer carácter del componente léxico a cuyo texto pertenecía el carácter sobre el que se pulsó el ratón. En un primer momento no hay ningún texto resaltado, puesto que aún no se ha insertado ningún carácter. A medida que el usuario va tecleando, el texto introducido se mostrará resaltado en los colores correspondientes a la operación de inserción. Se permite borrar texto que ha sido insertado, pero no borrar ni insertar fuera de los límites señalados por el texto resaltado.

Figura 7.7: Texto con operaciones de edición de componentes léxicos.



4. Una vez que se ha terminado de editar un componente léxico, si se desea realizar la misma operación sobre otro componente léxico, se repiten los pasos 2 y 3. En caso

de que se desee realizar un tipo de operación distinto sobre otro componente léxico, se debe repetir el proceso desde el paso 1.

Es conveniente realizar las siguiente aclaraciones sobre el comportamiento de ICEeditor:

- El texto de los componentes léxicos borrados permanecerá en pantalla hasta que se realice el siguiente análisis, resaltado en los colores correspondientes a la operación de borrado. Con ello se pretende mostrar al usuario una mejor perspectiva de las operaciones realizadas sobre los componentes léxicos.

- Una vez que se ha modificado un componente léxico, puede ser seleccionado posteriormente para volver a ser modificado. Sin embargo, el texto de las inserciones no puede ser modificado una vez que se ha dado por finalizada la operación de edición de componentes léxicos en la cual fue introducido. Ello se debe a que el nuevo componente léxico creado no existirá realmente hasta que se realice un nuevo análisis sintáctico.

- En la mayoría de las gramáticas, existen caracteres separadores de componentes léxicos. Al realizar la inserción de un nuevo componente léxico, debe tenerse la precaución de introducir un carácter que lo separe del componente léxico siguiente. Dicho carácter puede ser un punto y coma, un espacio, etc., dependiendo de la gramática. ICEeditor no introduce automáticamente este carácter porque se trata de una aplicación de carácter general, válida para cualquier tipo de gramáticas, y la utilización de caracteres separadores es una característica dependiente de la gramática que se esté utilizando en cada momento.

- Por la misma razón de la generalidad de ICEeditor, no se imponen restricciones sobre los caracteres que se pueden incluir en el texto. Por tanto, es posible introducir caracteres de retorno y avance de línea. En la figura 7.8 se muestra un texto de prueba en el que se han realizado operaciones de inserción y modificaciones en las cuales se han introducido caracteres de ruptura de líneas.

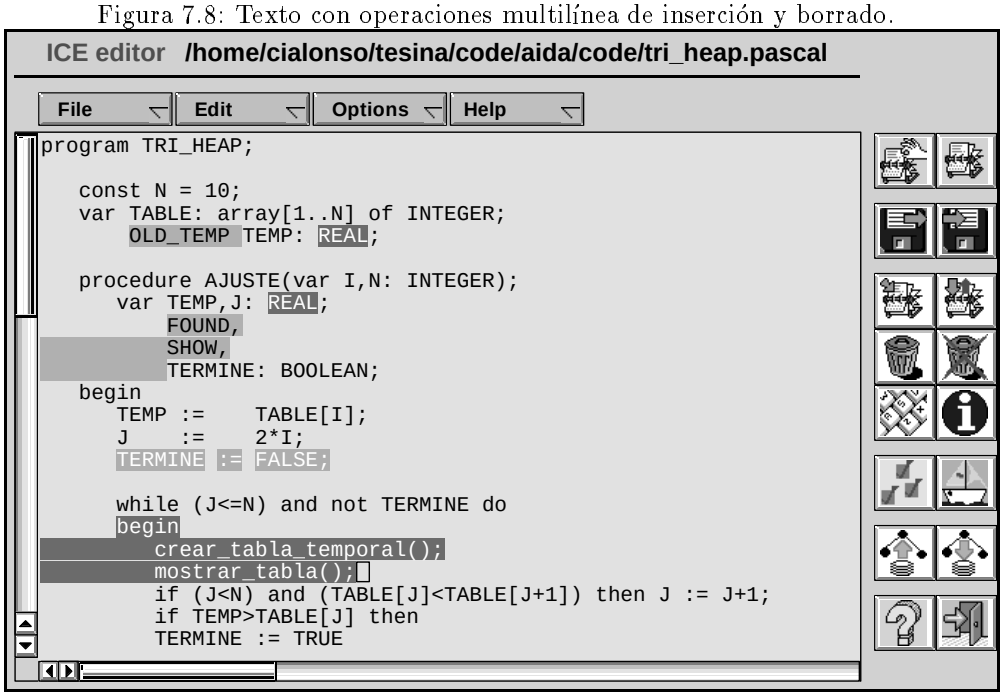
Independientemente de las modificaciones que se hayan realizado en el texto durante las operaciones de inserción y modificación, ICEeditor se encarga de mantener automáticamente la correspondencia entre los componentes léxicos y el texto.

7.3 El menú

Para facilitar la interacción del usuario con ICEeditor, se ha dispuesto un menú situado en la parte superior de la ventana principal. Este menú contiene un conjunto de submenús cuyos elementos permiten realizar casi todas las acciones disponibles.

Para seleccionar una opción del menú se debe pulsar con el botón izquierdo en la barra de menú, manteniéndolo pulsado. Inmediatamente se abrirá un submenú. Al ir arrastrando el cursor del ratón por el submeú se irán resaltando las distintas opciones. Cuando la opción que se desee escoger esté resaltada, se debe soltar el botón del ratón, con lo que se indica que se ha elegido dicha opción.

Figura 7.8: Texto con operaciones multilínea de inserción y borrado.

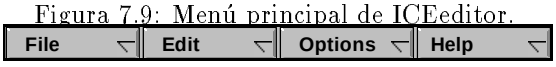


Aquellas opciones cuyo título termina en tres puntos indican que al ser seleccionadas se activará una ventana de diálogo. Aquellas opciones que tienen en su borde derecho un pequeño triángulo indican que de ellas cuelga un nuevo submenú. Cuando el ratón es arrastrado sobre una de estas opciones, se activa dicho submenú.

7.3.1 El menú principal

En la figura 7.9 se muestra el aspecto que presenta el menú principal de ICEeditor. Este menú está formado por una barra de botones. Al activar alguno de estos botones mediante el ratón, se desplegará un submenú.

Figura 7.9: Menú principal de ICEeditor.



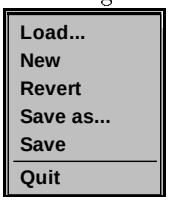
Las opciones que presenta el menú principal son las siguientes:

- *File*, que activa el submenú de gestión de ficheros, en el cual hay opciones que permiten manejar cómodamente los aspectos relacionados con la carga y grabación de archivos en el editor.
- *Edit*. Esta opción activa el submenú en el cual se agrupan las operaciones de edición de componentes léxicos.
- *Options* activa un submenú en el cual se pueden seleccionar distintas opciones, que se refieren tanto al análisis sintáctico como al aspecto de la aplicación ICEeditor.
- *Help* activa el submenú con las opciones de ayuda disponibles en ICEeditor.

7.3.2 El menú de gestión de ficheros

Este menú es realmente un submenú que cuelga del menú principal y que se activa al pulsar sobre la opción *file* de éste. Su misión consiste en facilitar todas aquellas labores relacionadas con el manejo de ficheros en ICEeditor. En la figura 7.10 se muestra el aspecto de este menú.

Figura 7.10: Menú de gestión de ficheros.



Las opciones seleccionables por el usuario son las siguientes:

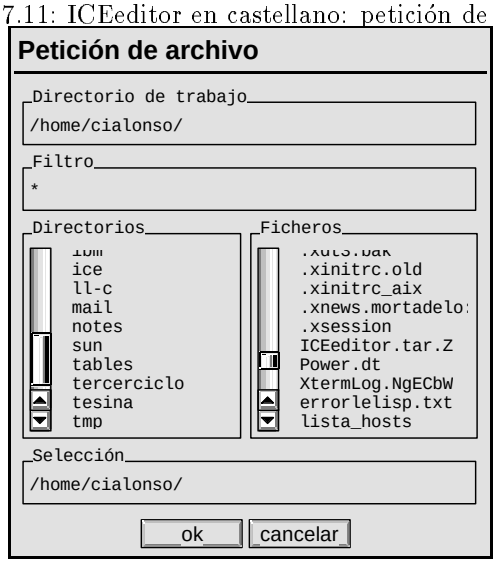
- *Load.* Al seleccionar esta opción se activa una ventana de diálogo como la que se muestra en la figura 7.11 mediante la cual el usuario puede seleccionar el fichero a cargar. Existen varios modos alternativos de realizar esta tarea:
  - El primero consiste en escribir directamente el nombre de fichero en el recuadro etiquetado *selection*. Esta es la única posibilidad si lo que se desea es cargar un fichero que no existe, es decir, si se desea crear un nuevo fichero.
  - Otro método consiste en seleccionar el nombre mediante la navegación por la estructura de directorios. Para ello disponemos de los paneles etiquetados *directories* y *files*. En el panel *directories* disponemos de una lista con los nombres de todos los directorios que cuelgan del directorio actual. Para facilitar el posicionamiento del usuario dentro del árbol de directorios, se muestra el directorio actual en el recuadro titulado *working directory*[9]. En el recuadro *files* se muestran los ficheros que contiene el directorio actual. Para facilitar la búsqueda en directorios que contienen un elevado número de ficheros, se dispone del recuadro titulado *filter*, en el cual se puede especificar un filtro para los nombres de ficheros utilizando las convenciones estándar Unix referentes al uso de expresiones regulares. En ambas cajas, *directories* y *files*, se dispone de barras de desplazamiento para facilitar la localización de los directorios y los ficheros.

  Una vez que se ha indicado convenientemente el nombre del fichero que se desea cargar, se debe pulsar el botón *OK* para confirmar realmente la operación de carga del fichero. En caso de que no se desee continuar con el proceso de carga, se dispone del botón *cancel*.

  Un intento del usuario de cargar un fichero cuando en el editor ya se encuentra otro sobre el que se han realizado algún tipo de cambios desde la última vez que fue escrito a disco, supone la pérdida de estas últimas modificaciones. Para prevenir situaciones indeseadas fruto del descuido del usuario, ICEeditor es capaz de reconocer tales

[9] Nada más aparecer la ventana de diálogo, el directorio actual se corresponde con el nombre del directorio desde el cual se lanzó ICEeditor. Su contenido va cambiando según el usuario va interactuando con los distintos elementos gráficos de esta ventana.

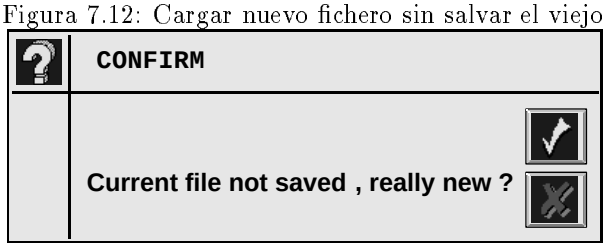
Figura 7.11: ICEeditor en castellano: petición de fichero.



situaciones presentando en estos casos una ventana de confirmación como la que se muestra en la figura 7.12. Cuando surge una de tales ventanas, el usuario tiene ante sí dos salidas posibles:

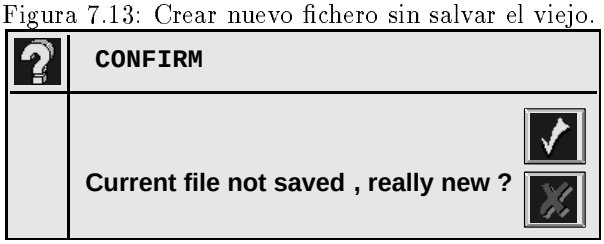
- Pulsar el botón superior, con lo cual confirma la realización de la acción, asumiendo los riesgos de pérdida de información que puedan surgir.
- Pulsar el botón inferior, que cancela la realización de la acción.

Figura 7.12: Cargar nuevo fichero sin salvar el viejo.



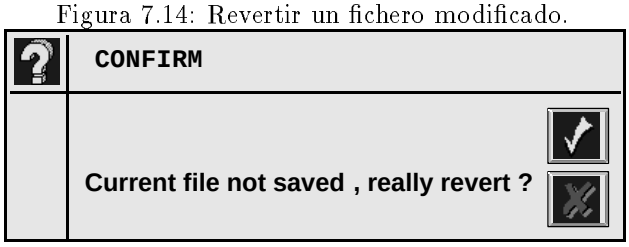
- *New.* Esta opción permite vaciar el contenido de ICEeditor, tanto del editor como de las estructuras asociadas al proceso de análisis léxico-sintáctico. Se usa principalmente cuando se desea utilizar el editor incorporado en ICEeditor para editar un nuevo fichero. Un intento de utilizar esta opción cuando ya se encuentra cargado un fichero con datos no actualizados en disco supondrá la aparición en pantalla de la ventana de advertencia que se muestra en la figura 7.13, cuyo funcionamiento es similar a la que surge en condiciones análogas en el caso de la opción *load* de este mismo menú.
- *Revert.* Permite restaurar el fichero que se encuentra en el editor a su estado inicial. Esto consiste en releer el fichero desde disco. Esta operación es equivalente

Figura 7.13: Crear nuevo fichero sin salvar el viejo.



a realizar la carga del mismo fichero que se está editando, pero evitando tener que seleccionar manualmente el nombre del mismo. Por precaución, se activa la ventana de confirmación que se muestra en la figura 7.14.

Figura 7.14: Revertir un fichero modificado.



- *Save as...* es la opción que se utiliza para renombrar un fichero. Más concretamente esta opción guarda en disco el contenido del editor, en un fichero suyo nombre es solicitado al usuario mediante la activación de una ventana de diálogo similar a la utilizada en el caso de la opción *load* de este mismo menú. Un método alternativo para renombrar un fichero consiste en utilizar el editor situado en la parte superior de ICEeditor para editar el nombre del fichero. Cuando se salve el fichero a disco utilizando la opción *save* del menú o el botón equivalente, se utilizará el nombre contenido en esa línea como nombre de fichero.
- *Save.* Opción utilizada para salvar el contenido de un fichero con el mismo nombre, reescribiendo el fichero original. Si se intenta grabar un fichero que no ha sufrido modificaciones con respecto a su versión residente en el disco, se muestra la ventana de confirmación de la figura 7.15.
- *Quit.* Esta opción permite salir de ICEeditor, finalizando la sesión de edición de componentes léxicos. Antes de salir realmente, si se han realizado cambios sobre el fichero que no han sido guardados efectivamente en disco, se muestra la ventana de advertencia que se muestrra en la figura 7.16.

7.3.3 El menú de edición

Una vez que se ha analizado un texto, se pueden realizar operaciones de edición sobre los componentes léxicos reconocidos. Para seleccionar la operación que se va a utilizar, el

Figura 7.15: Salvar un fichero sin modificaciones.

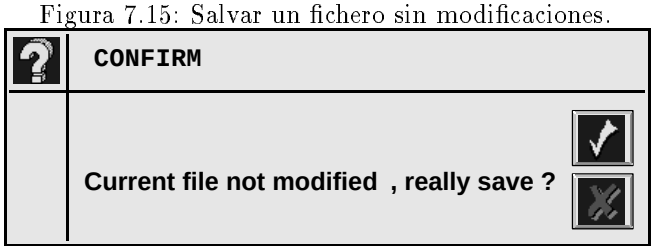
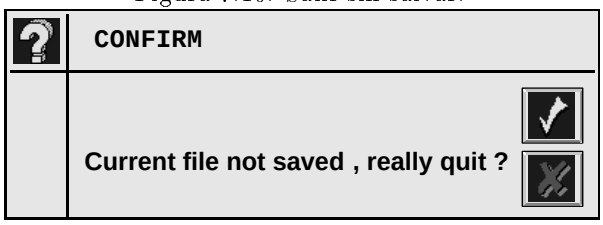
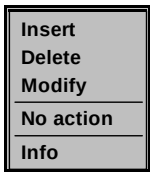


Figura 7.16: Salir sin salvar.



usuario dispone de un menú denominado *edit*[10] en el cual se encuentran todas las posibles operaciones que puede realizar. Este menú se muestra en la figura 7.17.

Figura 7.17: Menú de edición.



Las opciones del menú son las siguientes:

- *Insert.* Mediante la selección de esta opción el usuario establece que las siguientes operaciones de edición de componentes léxicos que se realicen serán operaciones de inserción. Esto conlleva que cuando se pulse con el ratón sobre un carácter del texto, ICEeditor permitirá insertar un componente léxico antes de aquel a cuyo texto pertenece el caracter señalado. El usuario advierte que está en modo inserción mediante el aspecto del cursor, ya que éste adquiere forma de lápiz cuando ICEeditor se encuentra en modo de inserción de componentes léxicos.

- *Modify.* Al seleccionar esta opción ICEeditor entra en modo de modificación, por lo que entenderá que todas las operaciones que se realicen a partir de ese momento sobre algún componente léxico serán operaciones de modificación. Mientras se encuentre en este modo de edición, el cursor tendrá la forma de dos flechas en posición de intercambio.

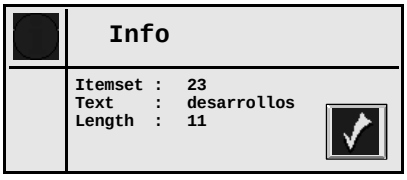
[10] El segundo botón de la barra de menú principal.

- *Delete.* Esta opción, cuando se selecciona, activa el modo de borrado de componentes léxicos en el editor. Esto quiere decir que cuando se pulse el botón izquierdo del ratón sobre algun carácter del texto, ICEeditor marcará automáticamente todo el texto del componente léxico como borrado. El texto desaparecerá realmente de la pantalla cuando se proceda a reanalizar el texto. Para que el usuario pueda advertir fácilmente que se encuentra en este modo de edición, el cursor adquiere la forma de una pequeña calabera.

- *No action.* Esta opción desactiva momentáneamente la edición de componentes léxicos. Es decir, las pulsaciones del ratón sobre el texto no tendrán ningún efecto sobre ellos. No se permite modificar el contenido del editor. Esta opción es útil para navegar libremente por el texto y para realizar selecciones de partes del texto con el ratón. Cuando ICEeditor se encuentra en este modo, el cursor pasa a tener la forma habitual de la barra vertical.

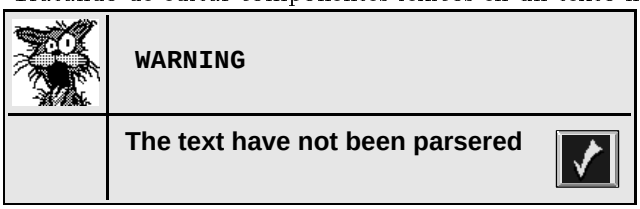
- *Info.* Al seleccionar esta opción el cursor pasará a tomar la forma de un signo de interrogación. Cuando el usuario haga click con el ratón sobre un carácter, se mostrará información sobre el componente léxico al cual pertenece dicho carácter. Esta información se muestra en una ventana de aspecto similar a la de la figura 7.18. Una vez que se haya visto la información debe pulsarse el botón para desactivar la ventana y poder seguir pidiendo información de otros componentes léxicos.

Figura 7.18: Ventana de información de un componente léxico.



Todas las operaciones mostradas anteriormente no se puden llevar a cabo si el texto no ha sido previamente analizado. En caso de intentar editar algun componente léxico en un texto sin analizar, ICEeditor hará aparecer la ventana que se muestra en la figura 7.19. Una vez leído el mensaje, se debe pulsar el botón de la parte derecha de la ventana para que ésta desaparezca.

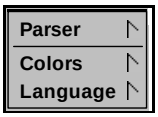
Figura 7.19: Tratando de editar componentes léxicos en un texto no analizado.



7.3.4 El menú de opciones

El usuario puede configurar a su gusto ciertos aspectos de ICEeditor, entre los cuales se incluye el color utilizado para resaltar el texto de los componentes léxicos editados y el idioma utilizado en la interacción con el usuario. Todas estas opciones de personalización está agrupadas en el menú *options* que se muestra en la figura 7.20. Como se puede observar, todas las opciones dan acceso a nuevos menús.

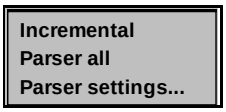
Figura 7.20: Menú de opciones.



El menú de parser

La opción *parser* del menú *options* da acceso al menú con las opciones de análisis sintáctico, que se muestra en la figura 7.21.

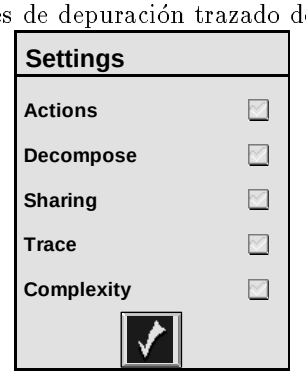
Figura 7.21: Menú de análisis sintáctico.



Este menú contiene tan tres opciones que el usuario pueda seleccionar:

- *Incremental.* Al seleccionar esta opción se realiza una análisis incremental del texto en el editor, tomando como base el análisis anterior y las operaciones realizadas sobre los componentes léxicos. Si el texto no había sido analizado previamente, entonces se realiza un análisis total.
- *Parser all.* Al seleccionar esta opción se realiza un análisis total del texto actualmente en el editor. No se tiene en cuenta la información obtenida de análisis previos ni de la edición de componentes léxicos, excepto claro está, el texto insertado y borrado.
- *Settings.* Al seleccionar esta opción aparecerá en pantalla la ventana que se muestra en la figura 7.22. Mediante pulsaciones del ratón se activan y desactivan las opciones de depuración y trazado del analizador sintáctico. Estas opciones son las siguientes:
  - *Actions.* Si se activa, se mostrarán las acciones de desplazamiento y reducción que tienen lugar durante el proceso de análisis.
  - *Decompose.* Muestra la descomposición de las operaciones de desplazamiento y reducción en las correspondientes operaciones de *push* y *pop* del autómata LALR(1).
  - *Share.* Su activación provoca que se indiquen, durante el análisis, los nodos compartidos del árbol de análisis.

Figura 7.22: Opciones de depuración trazado del análisis sintáctico..

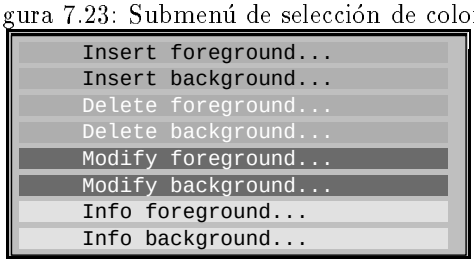


- *Trace.* Hace que durante el proceso de análisis se muestre una traza de la generación y expansión de los itemsets.
- *Complexity.* Cuando está activada, se muestra al final del análisis los valores de complejidad del mismo.

El menú de colores

El usuario puede adaptar a sus preferencias personales los colores utilizados por ICEeditor mediante las opciones presentes en el menú de colores que se muestra en la figura 7.23.

Figura 7.23: Submenú de selección de color.



Las opciones diponibles son las siguientes:

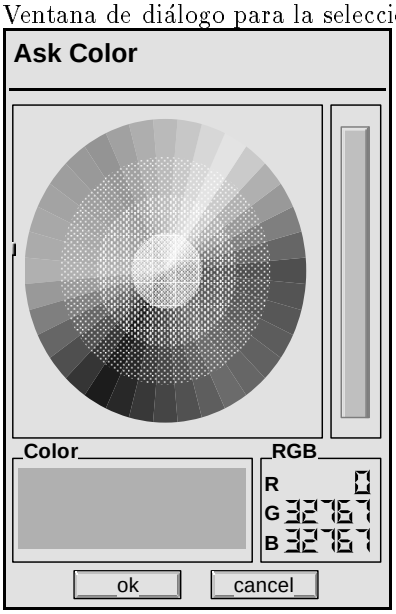
- *Insert foreground*, que permite cambiar el color de los caracteres pertenecientes a los nuevos componentes léxicos insertados.
- *Insert background*, que permite cambiar el color del fondo en los componentes léxicos insertados.
- *Delete foreground*, opción que establece el color utilizado en los caracteres que pertenecen a componentes léxicos borrados.
- *Delete background*, opción para modificar el color del fondo sobre el que se muestran los componentes léxicos borrados.
- *Modify foreground*, para cambiar el color de los caracteres en los componentes léxicos afectados por operaciones de modificación.
- *Modify background*, para cambiar el color del fondo en los componentes léxicos modificados.

- *Info foreground.* Con esta opción el usuario puede cambiar el color en el que se muestran los caracteres que pertenecen al texto del componente léxico sobre el que se solicita información.

- *Info background.* Permite cambiar el color del fondo utilizado para resaltar el componente léxico del cual se solicita información.

Todas las opciones terminan en puntos suspensivos. Esto quiere decir que al seleccionar cualquiera de ellas aparecerá una ventana de diálogo como la que se muestra en la figura 7.24.

Figura 7.24: Ventana de diálogo para la selección de color.



Esta ventana contiene una *rueda de color*[11] con todo el espectro de colores diponibles. Al pulsar el cursor dentro de esa rueda de color, en el recuadro de la parte inferior a la izquierda se verá una muestra del color exacto sobre el cual se ha pulsado, mientras que en el recuadro de la parte inferior a la derecha se muestran los valores de los componentes RGB[12] de dicho color. Si se arrastra el ratón con el botón pulsado, en dicho recuadro se irán mostrando los colores de los puntos por los que va pasando el ratón, mientras van cambiando los valores RGB en el recuadro de la derecha. Mediante la barra situada al lado derecho, en posición vertical, se puede regular la intensidad del color. Para ello debemos pulsar con el ratón a la altura deseada[13]. Se puede variar dinámicamente el brillo de un determinado color arrastrando el cursor a lo largo de la barra vertical.

Una vez que el usuario ha encontrado un color que es de su agrado, puede indicar que desea que sea utilizado pulsando el botón *OK*. Si por el contrario, no queda satisfecho con ningún color o simplemente prefiere los que ya están establecidos, puede pulsar el botón *cancel* para anular la selección de color.

[11] Color wheel.
[12] Red, Green, Blue.
[13] La barra representa una escala en la que el tope superior indica el 100% del brillo y el tope inferior ausencia total de brillo.

El menú de idioma

ICEeditor es capaz de utilizar cuatro idiomas distintos en el proceso de interacción con el usuario. Estos idiomas son el inglés, el francés, el español y el gallego. El usuario puede elegir el que más le convenga utilizando las opciones del menú de idioma. A este menú, que se muestra en la figura 7.25, se accede a través de la opción *language* del menú de opciones.

Como se puede ver el la figura 7.25, cada una de las opciones muestra en pantalla la bandera de un país representaivo del idioma, así como el nombre que el idioma recibe en su propia lengua.

Figura 7.25: Submenú de selección de idioma.

Al seleccionar una de estas opciones, todos los mensajes mostrados al usuario, incluyendo los rótulos de los menús y de sus opciones, los textos de las ventanas y de sus botones y paneles así como los mensajes de error, se muestran en el idioma elegido. Como ejemplo, en la figura 7.26 se muestra la pantalla principal de ICEeditor en un momento en el que se está utilizando el gallego como idioma actual. Por otra parte, en la figura 7.11 se muestra el contenido de una ventana de diálogo, en este caso la utilizada para pedir el nombre de un fichero, cuando el idioma utilizado es el castellano.

7.3.5 El menú de ayuda

Cuando el usuario no recuerda como realizar una determinada opción o cual es la combinación de teclas adecuada en el editor para hacer tal operación, debe recurrir a la ayuda incorporada en ICEeditor. Para ello se utiliza el menú *help*, que se muestra en la figura 7.27.

Este menú tiene tres opciones:

- *About ICEeditor* permite obtener información acerca del autor de la aplicación y de la versión de ésta que se está utilizando.

- *Help on ICEeditor* muestra una ventana de ayuda sobre el uso general de ICEeditor, centrándose en las operaciones con ficheros, el análisis léxico-sintáctico y la edición de componentes léxicos.

- *Help on editor* muestra una ventana de ayuda para el editor de textos incorporado en ICEeditor en la cual se indican todas las posibles operaciones que se pueden realizar en él, así como las combinaciones de teclas que se deben utilizar.

Figura 7.26: ICEeditor en gallego.

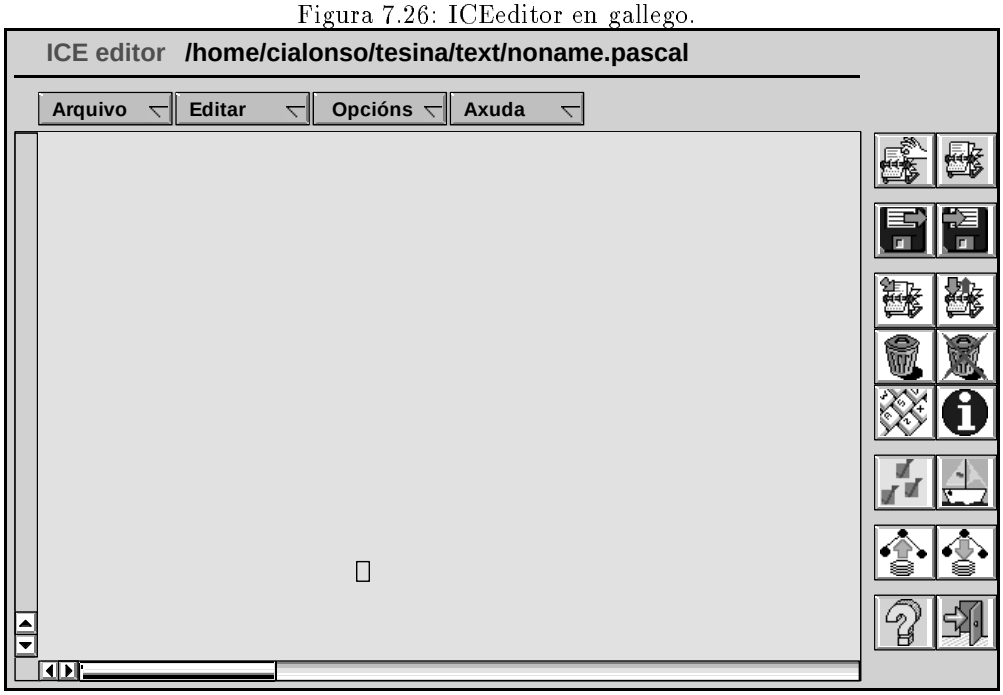
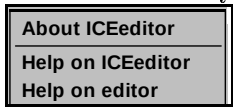


Figura 7.27: Menú de ayuda.



7.4 La barra de botones

Mediante la utilización del menú se pueden acceder a todas las opciones disponibles en ICEeditor. Para facilitar el acceso a ciertas características muy utilizdas el usuario dispone de una barra de botones situada junto al borde derecho de la ventana principal de ICEeditor. Mediante la pulsación de uno de estos botones se realizan ciertas opciones directamente, sin tener que navegar por los menús y submenús, con lo cual se gana en comodidad y en rapidez. Con la utilización de iconos descriptivos en cada botón se ha pretendido que el usuario sea capaz de reconocer, de un sólo golpe de vista, la función que realiza cada botón. Concretamente, la barra de botones está constituida por los siguientes botones:

 La pulsación de este botón conlleva la activación del proceso de parser incremental, tal y como sucedía al seleccional la opción *incremental parser* del menú de análisis sintáctico.

 Realiza un análisis total del texto. Equivalente a la opción *parser all* del menú de análisis sintáctico.

 Este botón permite cargar un nuevo fichero en el editor. El efecto es el mismo que seleccionar la opción *load* del menú de gestión de ficheros.

 Este botón permite salvar en un fichero en disco el contenido del editor. El resultado conseguido es equivalente al obtenido mediante la selección de la opción *save* del menú de gestión de ficheros.

 Al pulsar este botón se pone a ICEeditor en modo de inserción, en lo que respecta a la edición de componentes léxicos. Es equivalente a seleccionar la opción *insert* del menú de edición.

 Establece el modo de modificación en las operaciones con componentes léxicos, al igual que ocurría al seleccionar *modify* en el menú de edición.

 Activa el modo de borrado de componentes léxicos, lo mismo que sucedía al seleccionar *delete* en el menú de edición.

 Activa el modo de *desborrado* de componentes léxicos, lo mismo que sucedía al seleccionar *undelete* en el menú de edición.

 Con este botón se puede acceder al modo de no-acción de ICEeditor. Actúa igual que si hubiésemos pulsado sobre la opción *no action* del menú de edición.

 Mediante la pulsación de este botón se entra en el modo de información del editor de componentes léxicos, lo cual permite obtener información relativa a uno de los componentes léxicos . Su funcionalidad es la misma que la proporcionada por la opción *info* del menú de edición.


Con este botón se activa la ventana mediante la cual se establecen las opciones de trazado y depuración del analizador sintáctico. Actúa igual que si hubiésemos pulsado sobre la opción *settings* del menú de las opciones de análisis sintáctico.


Mediante la pulsación de este botón se activa la ventana de navegación que se muestra en la figura 7.29.


Mediante este botón se puede cargar una estructura de análisis previamente salvada a disco. El fichero que se busca tiene el mismo nombre que el fichero que se está editando, añadiéndole la extensión `.items`.


Con este botón se puede salvar a disco la estructura de análisis generada durante el proceso de análisis sintáctico. Dicha estructura se guarda en un fichero cuyo nombre coincide con el del fichero que se encuentra en el editor más la extensión `.items`. de no-acción de ICEeditor. Actúa igual que si hubiésemos pulsado sobre la opción *no action* del menú de edición.

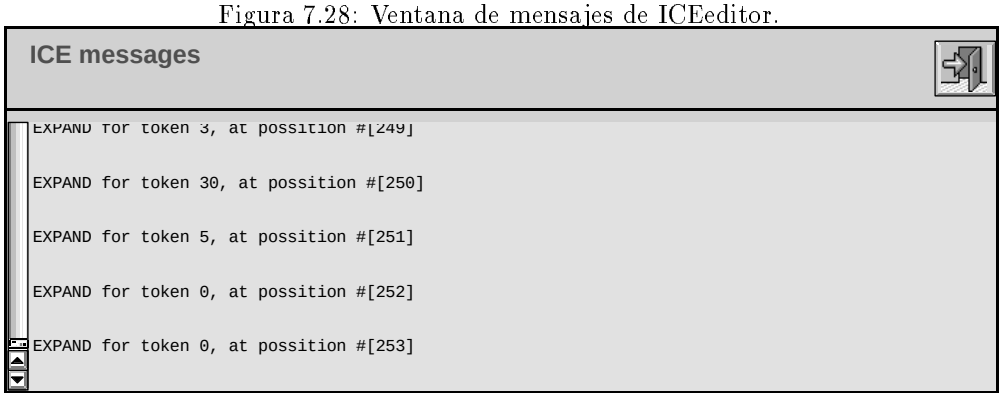

Da acceso a la ayuda general de ICEeditor, al igual que la opción *help on ICEeditor* del menú de ayuda.


Es el botón que permite terminar la sesión con ICEeditor. Es equivalente a seleccionar *quit* en el menú de ficheros.

7.5 La ventana de mensajes

Cuando se realiza un análisis sintáctico total del texto que se encuentra en el editor, surge la ventana de mensajes de ICEeditor, cuyo aspecto general se muestra en la figura 7.28. En esta ventana aparecerán todos los mensajes generados por el analizador. Los mensajes generados en los sucesivos análisis incrementales que se realicen a partir de ese momento serán mostrados también en esta ventana, a continuación de los correspondientes al análisis previo.

Figura 7.28: Ventana de mensajes de ICEeditor.



Esta ventana consta de tres componentes:

- Un editor en el cual van apareciendo los mensajes a medida que van siendo generados por el analizador sintáctico. No es posible editar el contenido de este editor.
- Una barra de desplazamiento mediante la cual es posible observar todos los mensajes producidos durante el análisis.
-  el botón mediante el cual se puede cerrar la ventana de mensajes.

La ventana de mensajes permanece activa hasta la realización del siguiente análisis total, en cuyo caso es sustituida por una nueva, o hasta que se carga un nuevo fichero en el editor. Al salir de la ventana principal de ICEeditor, la ventana de mensajes es eliminada automáticamente.

7.6 La ventana de navegación

Al pulsar el botón *navigate* ubicado en la barra de botones de la ventana principal de ICEeditor, se tiene acceso a la ventana de navegación que se muestra en la figura 7.29. Nada más activarse, en esta ventana se muestra el nodo raíz del bosque compartido generado por el análisis. Mediante la utilización de los controles incluidos en esta ventana es posible navegar por la estructura arborescente.

Esta ventana esta constituida por los siguientes elementos:

- Un editor en el cual se muestran los nodos del bosque compartido por los cuales se va pasando durante la navegación. El texto mostrado en este editor no puede ser modificado.
- Una barra de desplazamiento que permite observar la representación de nodos por los que ya se ha pasado.
- Los controles de navegación, situados en la parte superior derecha de la ventana. Estos controles están formados por el siguiente conjunto de botones:

   Permite acceder al nodo padre del nodo actual. En el caso de que el nodo en que nos encontremos sea un nodo compartido, el nodo padre que se considerará será aquel desde el cual se descendió al nodo actual. Si se trata de acceder al padre del nodo raíz se obtendrá un mensaje de error.

   Permite acceder al primer hijo del nodo actual.

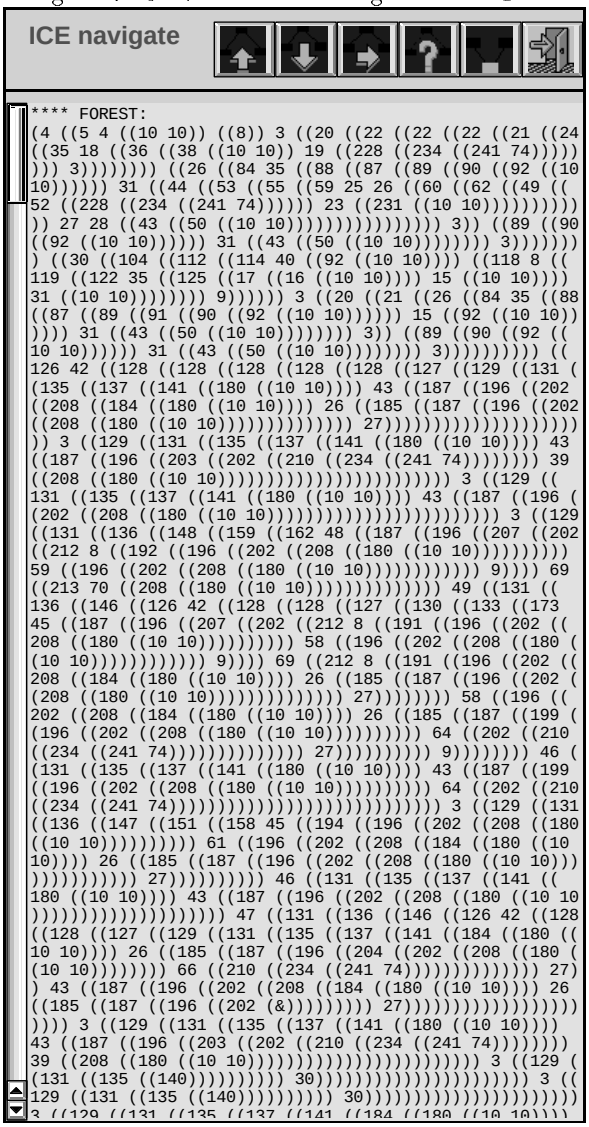
   Permite acceder al primer hermano del nodo actual, o lo que es lo mismo, al siguiente hijo del padre del nodo actual.

   Cuando el nodo actual representa una de las ramas opcionales del análisis [14], este botón permite acceder a la sigiente opción.

   Permite acceder a la representación de los nodos compartidos del bosque de análisis.

[14] Estas ramas opcionales son consecuencia de las ambigüedades de la gramática. El carácter no determinista del analizador ICE permite optener todas las posibles alternativas cuando surgen ambigüedades.

Figura 7.29: Ventana de navegación de ICEeditor.



 Cierra la ventana de navegación de ICEeditor. Esta ventana también es cerrada automáticamente cuando se finaliza la sesión con ICEeditor.

Capítulo 8

ICEeditor según Centaur

En este capítulo se muestra un boceto de otra posible implementación de ICEeditor. En este caso se ha utilizado la herramienta CENTAUR, desarrollada en el INRIA de Sophia-Antipolis (Francia) con el fin de proporcionar un conjunto de formalismos para el desarrollo de parsers. En el apéndice C se enumeran los diferentes componentes de CENTAUR y se realiza una breve descripción del componente gráfico *gfxobj*, que permite la construcción de interfaces gráficas simples que interactúen con los demás componentes del sistema.

8.1 Iniciando CENTAUR

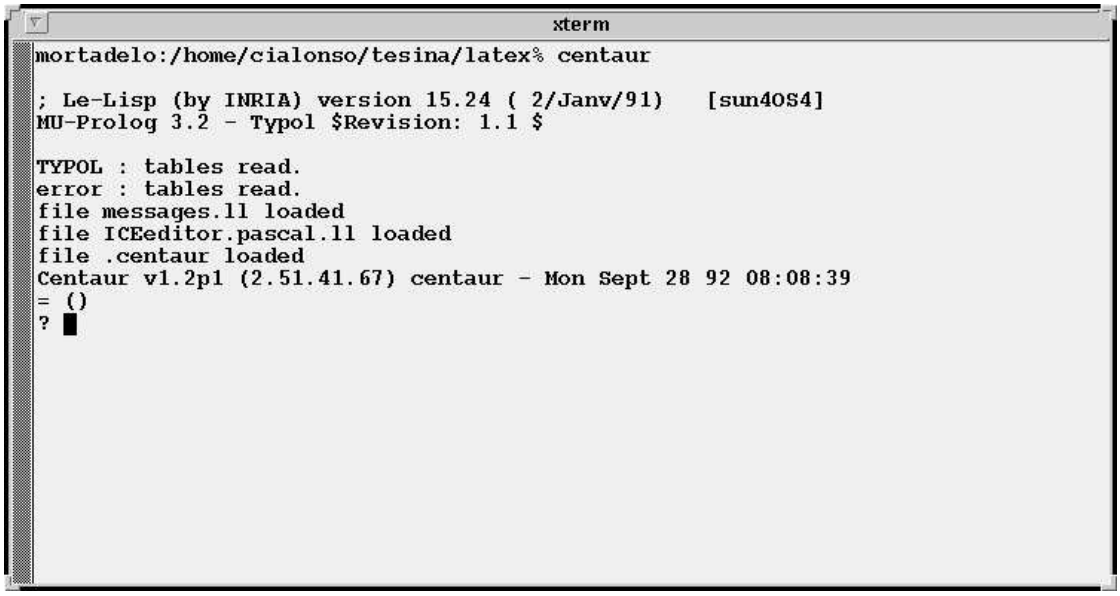
Una vez que el sistema CENTAUR ha sido correctamente instalado en la máquina[1], se puede arrancar tecleando `centaur` desde cualquier directorio. En la figura 8.1 se muestran los mensajes que aparecen en un `xterm` como consecuencia del arranque del sistema. En ellos se nos informa de la versión de LE-LISP que está siendo utilizada, de las tablas que han sido leídas, de la versión de Mu-Prolog utilizada con TYPOL y de información adidicional generada por los ficheros de configuración. Esta ventana también será utilizada para mostrar los diferentes mensajes que se generen durante la sesión de trabajo.

Una vez que ha sido arrancado CENTAUR, aparece el prompt de LE-LISP. La versión que se incluye de este lenguaje es completa, por lo que se puede utilizar para evaluar cualquier programa escrito en LE-LISP[2]. Llamando a la función `end`, que no toma argumentos, se termina la sesión con CENTAUR. Para acceder al lenguaje Prolog es preciso realizar una llamada a la función `prolog`, que tampoco toma argumentos. Para salir del entorno Prolog y regresar al entorno LE-LISP, debe pulsarse la combinación de teclas <CONTROL>D.

[1] Actualmente sólo se soportan sistemas Sun SPARC con SunOS 4.1.x, estaciones DEC bajo Ultrix y estaciones Silicon Graphics bajo IRIX. En todos los casos se debe disponer del X Window System.

[2] Para evitar la duplicidad de software instalado y por consiguiente ahorrar espacio en disco, en vez de instalar el LE-LISP incluido en la cinta de distribución de CENTAUR se puede utilizar otra versión previamente instalada, por ejemplo la que viene incluida con AÏDA y MASAÏ. Sin embargo, no es posible ejecutar programas construidos mediante estas últimas herramientas empleando CENTAUR, ya que las librerías de objetos y las ampliaciones de LE-LISP que utilizan no son accesibles desde fuera de sus entornos de desarrollo y de los módulos ejecutables generados mediante el runtime correspondiente.

Figura 8.1: `xterm` desde el que se lanzó CENTAUR.



8.1.1 Los ficheros de configuración

Cuando CENTAUR inicia su ejecución, lee un conjunto que ficheros de configuración que establecen ciertas características del entorno, que tendrán efecto hasta el fin de la sesión a menos que sean modificadas explícitamente en el transcurso de la misma.

El fichero `.lelisp`

Este es el fichero de configuración del LE-LISP que permite a cada usuario personalizar las características del entorno. Está situado en el directorio *home* del usuario. Un aspecto muy importante a tener en cuenta es que este fichero será leido siempre que se inicie una sesión con LE-LISP. Esto incluye todas las sesiones de AÏDA, MASAÏ y CENTAUR. Por tanto, en este fichero se deben establecer las caracteísticas de configuración generales para todas las sesiones LE-LISP. Para establecer aspectos concretos de cada herramienta se deben utilizar los ficheros de configuración específicos correspondientes a cada una de ellas.

El fichero `.centaur`

Este fichero contiene expresiones LE-LISP que establecen las características del entorno de trabajo específicas para CENTAUR. Como norma genral, es conveniente que cada fichero de cofiguración finalice con una intrucción que informe por pantalla de la lectura del fichero. Por ejemplo, en este caso se utilizaría una expresión como `(print "file .centaur loaded")`.

El fichero `.CENTAUR_SRC`

No se trata realmente de un fichero, sino de un enlace sinbólico. Su utiliación no es obligatoria, pero sí recomendable. Se utiliza principalmente para indicar la localización de ciertos ficheros y directorios en las bases de datos de recursos. Debe estar situado en el directorio home del usuario. Para crearlo se utiliza la siguiente instrucción Unix:

```
ln -s <directorio-raiz-de-centaur> .CENTAUR_SRC
```

donde <`directorio-raiz-de-centaur`> indica el directorio bajo el cual se ha instalado el contenido de la cinta de distribución CENTAUR, generalmente `/usr/local/centaur`.

8.1.2 Los ficheros de recursos

En la versión 1.2 de CENTAUR[3] aparecen por vez primera los ficheros de recursos, en los cuales se especifica la información de configuración de cada uno de los lenguajes utilizados. Estos ficheros, creados tomando como inspiración los ficheros de recursos existentes en el X Window System, se organizan en una jerarquía, de modo que a partir de un fichero raiz denominado `.centaur.rdb` se indican los ficheros de recursos de cada lenguaje definido, en los cuales a su vez se indica la ubicación de ficheros de recursos para cada una de los formalismos específicos utilizados.

El fichero `.centaur.rdb` se encuentra ubicado en el directorio home del usuario. En él se indican los diferentes lenguajes que se van a utilizar, su localización y el fichero de recursos corespondiente. Por ejemplo, para declarar la existencia de un lenguaje Pascal, el contenido de este fichero debería ser:

```
Centaur.UserFormalism: (pascal)
Centaur.pascal.Database.UserDefaults: centaur/tables/pascal/syntax/pascal.rdb
```

La primera línea declara los diferentes lenguajes definidos por el usuario, mientras que la segunda indica dónde se encuentra la base de datos de recursos[4] para ese lenguaje. Mediante `UserDefaults` se indica que la jerarquía de directorios utilizada comienza en el directorio home del usuario. Nótese la similitud del formato de los recursos con el utilizado por el X Window System.

8.1.3 La ventana principal de CENTAUR

Al iniciar la sesión con CENTAUR, se muestra en pantalla una pequeña ventana como la que se muestra en la figura 8.2[5].

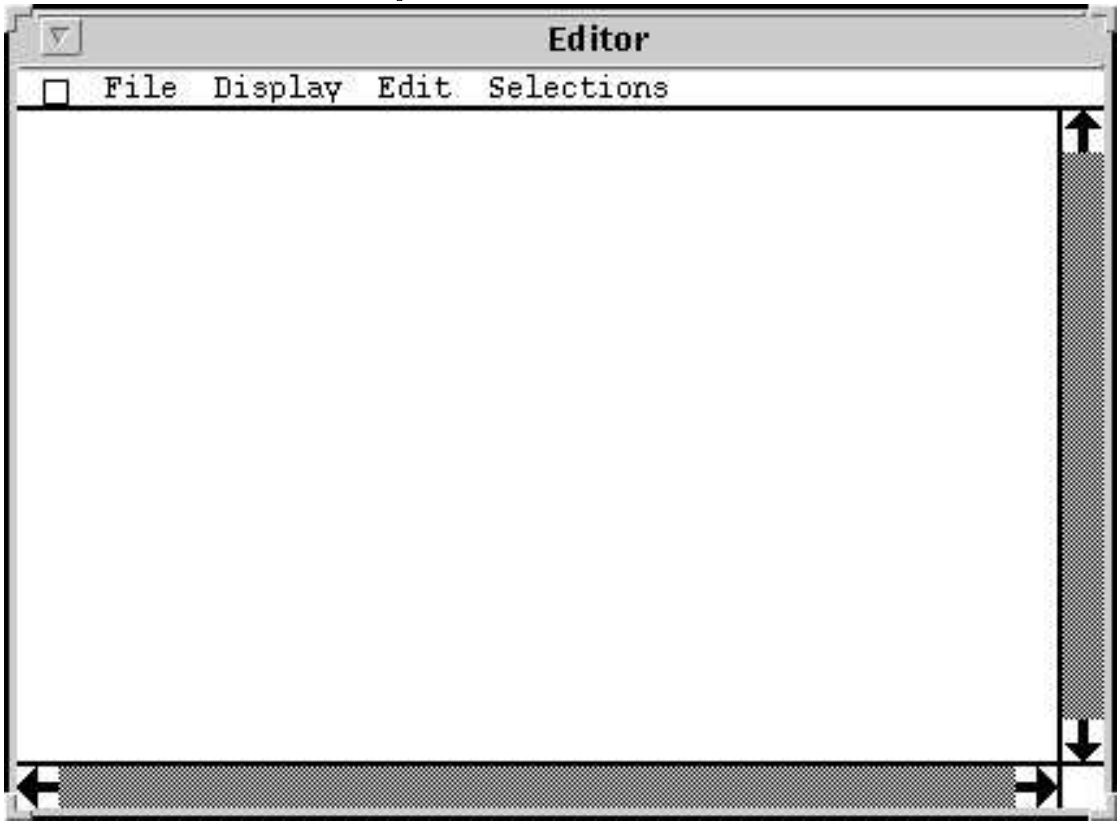
Desde esta ventana se puede acceder a diversas funcionalidades de CENTAUR, entre las cuales destaca la posibilidad de activar múltiples editores para los diferentes formalismos. El significado de cada uno de uno de los botones que la componen es el siguiente:

- *Editor*. Crea un nuevo editor para visualizar y manipular objetos estructurados. Estos editores presentan inicialmente el mismo aspecto que el de la figura 8.3. Una vez que se carga un programa, su aspecto se adapta convenientemente al formalismo que se está utilizando.

[3] La última por el momento, aunque se espera que pronto salga a la luz la versión 2.0, que promete numerosas mejoras.
[4] El fichero de recursos.
[5] El tamaño de esta ventana no puede ser modificado, ya que ignora los eventos de redimensionamiento.

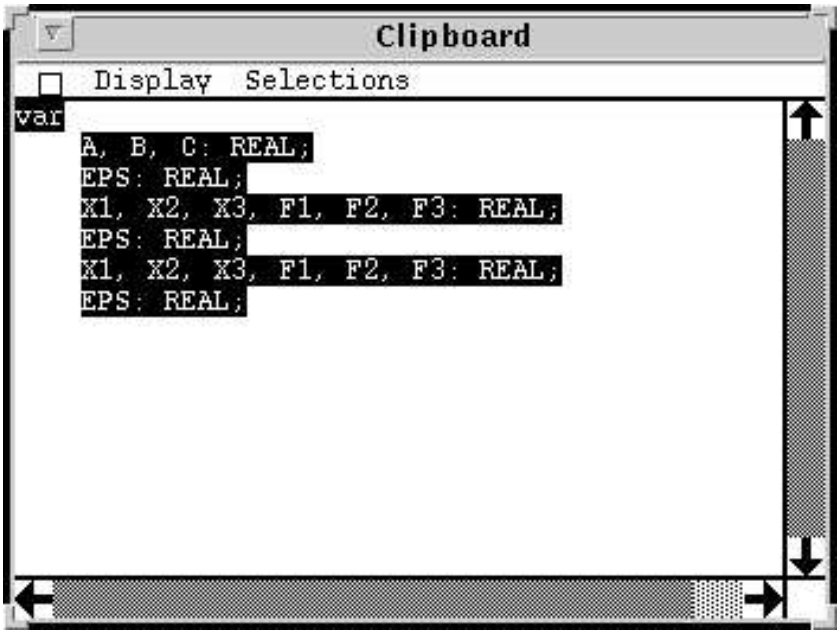
Figura 8.2: Ventana principal de CENTAUR.

Figura 8.3: Editor de CENTAUR.



- *Clipboard.* Crea o, si ya existe, hace visible el clipboard, que es un editor especial utilizado por los comandos de edición. De modo similar a como actúan las operaciones de cortar y pegar en la mayoría de las aplicaciones actuales, cuando desde un editor se corta una porción de la imagen mostrada, ésta es traspasada al clipboard. En la figura 8.4 se muestra un clipboard que contiene el objeto correspondiente a la parte de declaración de variables de un programa en Pascal.

Figura 8.4: Contenido del clipboard.

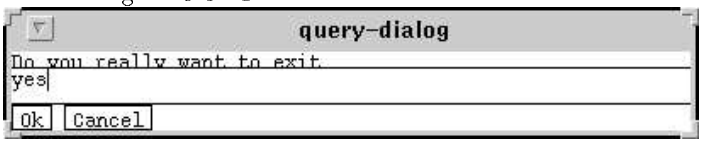


- *Reset resources.* Provoca la relectura de todos los ficheros de recursos, con lo que se actualiza la información global de recursos. Se debe utilizar siempre después de cada modificación en alguno de dichos ficheros, pues de lo contrario los cambios realizados no tendrán efecto en la sesión actual. Las herramientas que estén activas no variarán su aspecto inmediatamente. Para que ello suceda deberán ser canceladas y abiertas de nuevo.

- *Gcinfo.* Muestra información sobre la última operación de *garbage collection* llevada a cabo por LE-LISP.

- *End.* Finaliza la sesión con CENTAUR, solicitando confirmación al usuario mediante la ventana de diálogo que se muestra en la figura 8.5.

Figura 8.5: Confirmación de final de sesión.



8.2 Los formalismos de especificación

A continuación se va realizar una breve descripción de algunos de los formalismos disponibles en el entorno CENTAUR, que permiten la definición de la sintaxis abstracta y concreta de un lenguaje así como la forma en que será mostrado en los editores de CENTAUR.

Para mostrar cada uno de los componentes se utilizará como ejemplo la definición del lenguaje Pascal, ya que aún no está disponible una definición completa de un lenguaje natural en tales herramientas. La elección de un determinado lenguaje para ilustrar las funcionalidades de CENTAUR no supone ninguna limitación, ya que su forma de actuar es independiente del lenguaje.

8.2.1 El formalismo METAL

METAL es un formalismo que permite definir separadamente la sintaxis abstracta y concreta de un lenguaje. El uso de operadores de construcción de árboles permite controlar libremente la relación entre la sintaxis abstracta y la concreta.

Este formalismo utiliza Lex y Yacc para generar los analizadores léxico y sintáctico, respectivamente, por lo que las gramáticas de contexto libre definidas deberán ser de tipo LALR(1).

La sintaxis abstracta

La definición de la sintaxis abstracta delimita el conjunto de árboles abstractos que pueden ser construidos a partir del análisis sintáctico de los tokens. En la definición de la sintaxis abstracta se utilizan nodos, llamados *operadores*, y tipos de nodos, llamados *phyla*. Los operadores representan a los símbolos terminales y no terminales del lenguaje. Un operador viene definido por el número de hijos en el árbol y el tipo de tales hijos, esto es, el *phylum* al que pertenecen.

A continuación se muestra una pequeña porción de la sintaxis abstracta del Pascal:

```
abstract syntax
   ident   -> imlemented as IDENTIFIER ;
   intcst  -> implemented as INTEGER ;
   alfacst -> implemented as STRING ;
   charsct -> implemeted as CHAR ;
   .
   .
   .
   uminus  -> EXP ;
   uplus   -> EXP ;
   plus    -> EXP EXP ;
   minus   -> EXP EXP ;
   mult    -> EXP EXP ;
   div     -> EXP EXP ;
   mod     -> EXP EXP ;
   lident  -> IDENT + ... ;
   .
   .
   .
```

```
EXP    ::= ident intcst alfacst charcst nil hexcst realcst setof not
   uplus uminus unref hexa eql lss gtr  neq leq geq in intdiv
   mod div mult plus minus or and index dot call ;
IDENT  ::= ident ;
LIDENT ::= lident ;
.
.
.
```

Los nombres de los operadores se escriben en minúsculas, mientras que para los *phyla* se utilizan nombres en mayúsculas.

Los operadores del primer bloque son atómicos. La frase `implemented as` *type*, donde *type* indica un tipo LE-LISP, significa que cuando uno de tales operadores es reconocido por el analizador léxico es convertido inmediatamente en el tipo LE-LISP *type*.

Los operadores del segundo bloque `uminus` y `uplus` son unarios y su único descendiente en el árbol tiene por tipo al phylum `EXP`. El resto de los operadores mostrados en ese bloque son binarios, con ambos hijos de tipo `EXP`, excepto `lident`, que tiene uno o más descendientes de tipo `IDENT`[6].

En el tercer bloque se indica que el phylum `EXP` es el tipo de los operadores `ident`, `intcst`, `alfacst`, `charcst`, `uminos`, `uplus`, `plus`, `minus`, etc. También se indica que el phylum `IDENT` es el tipo del operador `ident`, que ya estaba incluido en el phylum `EXP`. Un operador puede pertenecer a más de un phylum, lo cual significa que puede aparecer en diferentes contextos.

La sintaxis concreta

Para definir la sintaxis concreta se utilizan reglas compuestas de dos partes:

- Una regla de producción que indica la sintaxis concreta. Son prácticamente iguales a las utilizadas en Yacc.
- Una función de construcción de árboles, que se utiliza para construir el árbol sintáctico en base a los terminales y a los no terminales instanciados.

A continuación se muestra un pequeño fragmento de la sintaxis concreta utilizada para definir el lenguaje Pascal:

```
rules
   <ID> ::= %ID ;
      ident-atom(%ID)
   <ID> ::= "pascal" ;
      ident-atom('pascal')
   <ID> ::= "forward" ;
      ident-atom('forward')
   <ID> ::= "external" ;
      ident-atom('external')
   <ID-LIST> ::= <ID> ;
      lident-list((<ID>))
   <ID-LIST> ::= <ID-LIST> "," <ID> ;
      lident-post(<ID-LIST>, <ID>)
```

[6] Si en vez de `+ ...` se hubiese escrito `* ...` significaría que `lident` podría tener 0 ó más descendientes de tipo `IDENT`.

Los no terminales se encierran entre los símbolos < y >. El carácter ; se utiliza para separar la regla de producción de la función de construcción de árboles.

Las funciones `list`, `pre` y `post` se utilizan respectivamente para crear una lista, para añadir un elemento al principio de una lista y para añadir un elemento al final de una lista. La función `list` requiere dos conjuntos de paréntesis para construir la lista.

Para construir nodos atómicos se utiliza la función `atom`. Por ejemplo, en la primera regla, `%ID` representa el token devuelto por el analizador léxico, mientras que `ident` representa el operador de la sintaxis abstracta cuyo valor atómico es el token almacenado en `%ID`.

El fichero de recursos

A la hora de definir un lenguaje, se recomienda que todos los ficheros referentes al mismo residan en una jerarquía de directorios que tenga como raiz `$HOME/centaur/tables/`*lang*, donde *lang* es el nombre del lenguaje que se va a implementar. En este directorio se deberá crear un subdirectorio `metal` que contendrá toda la información relativa a este formalismo. En el fichero de recursos del lenguaje *lang*, que en este caso será `$HOME/centaur/tables/syntax/pascal.rdb`, se indicará que Pascal es un formalismo del usuario, así como la localización de las tablas del lenguaje:

```
Centaur.pascal.Root: user
Centaur.pascal.Location: centaur/tables/pascal/syntax/metal
Centaur.pascal.Mode: std
```

El fichero `Buildfile`

Este fichero, localizado en `centaur/tables/pascal/syntax/metal`, contiene las directivas para la construcción del analizador sintáctico, para lo cual se hará uso de Yacc y Lex. El contenido de este fichero será el siguiente:

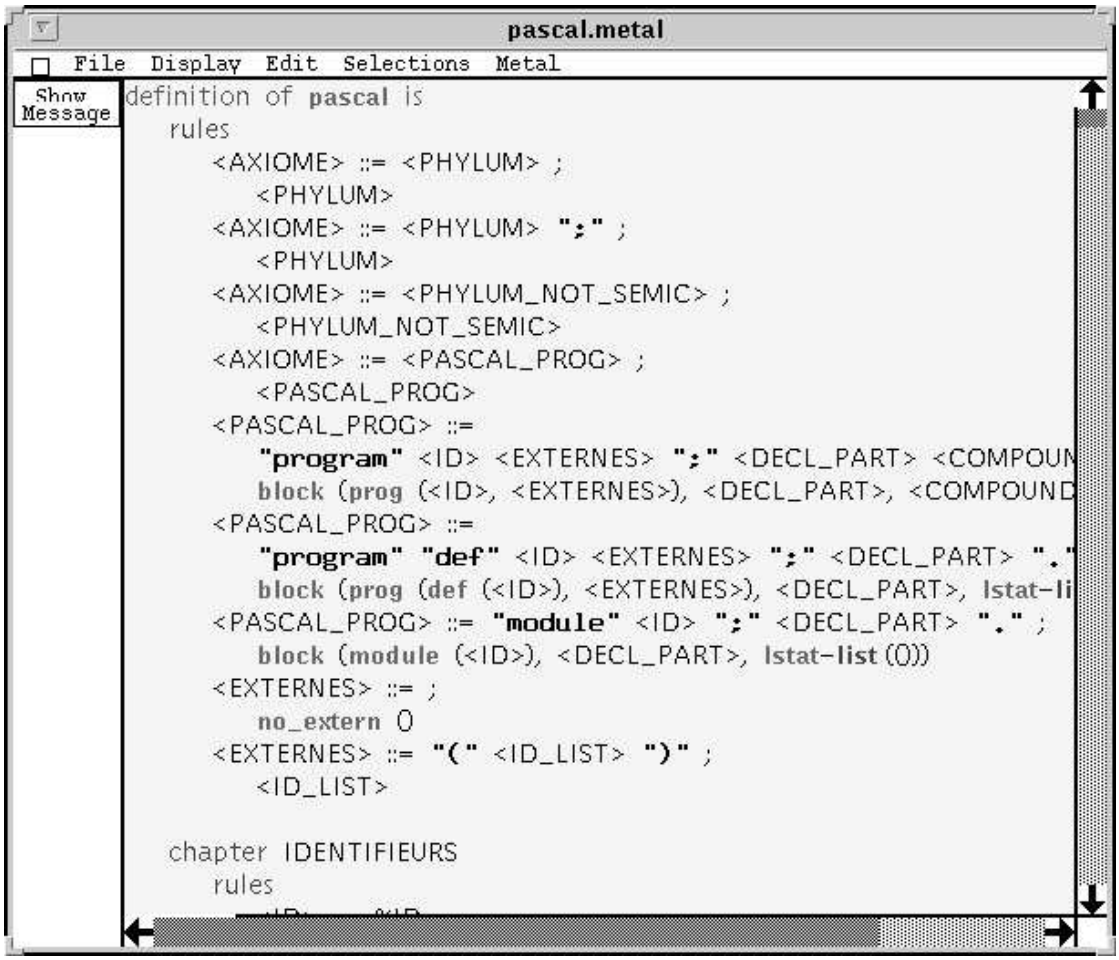
```
LANGUAGE=pascal
all: allmetal
#include /usr/local/centaur/tables/metal/Buildmetal
```

Al ejecutar el comando `ctmake` se construirá el parser. El compilador METAL creará el fichero `pascal.tokens.x`, que indica el tipo de tokens que tendrá que reconocer el analizador léxico. Se debe editar a mano este fichero para incluir las expresiones regulares de los tokens.

El editor CENTAUR para METAL

Se pueden editar programas METAL utilizando el editor de CENTAUR, accesible mediante el botón *Editor* de la ventana principal. Al introducir un nombre de fichero con extensión `.metal` en la ventana de diálogo, el editor resultante será un *ctview* adaptado especialmente para la edición de programas METAL. En la figura 8.6 se muestra un editor de este tipo utilizado para editar el fichero `pascal.metal`.

Figura 8.6: La especificación `pascal.metal` en el editor.



8.2.2 El formalismo SDF

Un formalismo alternativo para la definición de la sintaxis de un lenguaje es SDF[7]. Su principal diferencia con respecto a METAL es que permite analizar gramáticas de contexto libre arbitrarias. Una característica muy peculiar es que incluye las directivas para el análisis léxico.

Un programa SDF consta de varias secciones en las cuales se definen:

- *Sorts*, que es el nombre que reciben todos los no terminales en este formalismo.
- *Sintáxis léxica*, esto es, la definición léxica, utilizando expresiones regulares, de los terminales.
- *Syntaxis de contexto libre*. En cada regla se indica la sintaxis concreta, los phyla descendientes y el phylum obtenido como resultado por el operador. En las reglas también se puede indicar la asociatividad de los operadores.
- *Prioridades*, utilizadas para la desambiguación.
- *Variables*, concretamente esquemas de nombres para variables.

En la definición de Pascal no se ha utilizado este formalismo, optándose por METAL, que proporciona una mayor modularidad al separar los analizadores léxico y sintáctico.

8.2.3 El formalismo PPML

Este formalismo se utiliza para especificar la representación textual de un árbol de sintaxis abstracta. Un programa PPML[8] está constituido por reglas de la forma *patrón* `->` *formato*, en la que *patrón* indica los operadores de la sintaxis abstracta que reconoce una regla y *formato* describe su diposición en el formato de un *lenguaje de cajas*.

Se pueden escribir múltiples *pretty printers* para un lenguaje, pero uno de ellos tiene especial relevancia: es el llamado `std`, que debe respetar y reproducir la sintaxis concreta del lenguaje.

A continuación se muestra una regla del código PPML utilizada en la construcción del pretty printer `std` para Pascal:

```
plus(*exp1, *exp2) -> [<hv> if prec(*exp1) = 4 then "(" *exp1 ")"
    else *exp1
    end if
    "+"
    if prec(*exp2) >= 1 then "(" *exp2 ")"
    else *exp2
    end if];
```

El patrón de la regla indica que cuando en el árbol aparezca el operador `plus` con dos descendientes, se deberá formatear de acuerdo con lo indicado en la parte derecha de la regla. Las variables `*exp1` y `*exp2` son instanciadas con el árbol de los descendientes izquierdo y derecho, respectivamente.

En la parte de formato de la regla, `<hv>` se utiliza para indicar la concatenación horizontal de los elemntos siguientes hasta el final de la línea. las cajas que no quepan

[7] **S**yntax **D**efinition **F**ormalism.
[8] **P**retty **P**rinter **M**eta **L**anguage.

en la línea actual serán pasadas a la siguiente línea comenzando en el margen izquierdo marcado por la caja `hv` actual. Una caja puede contener terminales, variables y otras cajas.

La utilización del `if` para comparar prioridades permite determinar si la expresión definida por el operador `plus` deberá ser mostrada rodeada de paréntesis o no.

Los ficheros de recursos

Todos los pretty printers para un lenguaje deben ubicarse bajo el directorio `$HOME/centaur/tables/`*lang*`/pprinters`. En el fichero `pascal.rdb` debe añadirse la siguiente línea:

```
Centaur.pascal.ppml.Database.UserDefaults: centaur/tables/pascal/pprinters/pprinters.rdb
```

En el fichero `pprinters.rdb` se deberá identificar el fichero de recursos para `std`:

```
Centaur.pascal.ppml.std.Datrabase.Userdefaults: centaur/tables/pascal/pprinters/std/std.rdb
```

El fichero `Buildfile`

Para poder construir un módulo para `std`, debe existir un fichero `Buildfile` en el directorio `std`. Este fichero, en el caso del lenguaje Pascal, es el siguiente:

```
LANGUAGE=pascal
LANGDIR=./
ROOTDIR=../../../../../.CENTAUR_SRC/
USERDIR=../../../../../
MODNAME=cenatur/tables/pascal/pprinters/std
PPRINTER=std
ATFILE=
PPMLOPTIONS='module
#include ../../../../../.CENTAUR_SRC/centaur/tables/ppml/BuoldppmlLM
```

donde:

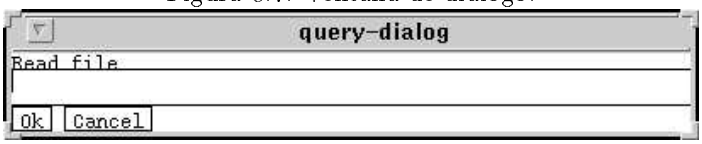
- `LANGDIR` indica la localización de las tablas del lenguaje.
- `USERDIR` indica la localización del directorio home del usuario.
- `MODNAME` es el directorio del módulo para el pretty printer.
- `PPRINTER` es el nombre del pretty printer.
- `ATFILE` indica si el fichero `pascal-std.at` es leido o no durante la compilación. En este caso, al no indicarse nada, no se lee. Este fichero se utiliza para definir funciones LE-LISP que pueden ser referenciadas dentro del programa PPML.
- `PPMLOPTIONS` indica al compilador PPML que debe crear un módulo compilado.

Todos los directorios son relativos al directorio en el que está almacenado el fichero `Buildfile`. La ejecución del comando `ctmake` provocará la compilación del pretty printer.

8.3 El entorno del lenguaje: ICEeditor

Cuando se pulsa el botón *Editor* de la ventana principal de CENTAUR, aparece un *cteview*, básicamente una ventana que contiene un editor denominado *ctedit* y unos elementos opcionales: un menú, una barra de botones y un scroller. Al pulsar el botón *Read*, aparecerá la ventana de diálogo que se muestra en la figura 8.7. Una vez introducido el nombre de un programa, éste será cargado en el *ctview* junto con su entorno asociado. El entorno de un lenguaje puede alterar la composición del menú, de la barra de botones y de las barras de desplazamiento, así como el comportamiento del ratón.

Figura 8.7: Ventana de diálogo.



En el caso del Pascal, el entorno se utiliza para lograr convertir el *ctedit* en la versión CENTAUR de la parte gráfica de ICEeditor.

Se deben incluir las siguientes líneas en el fichero `pascal.rdb` para indicar la existencia y ubicación de la especificación del entorno:

```
Centaur.pascal.Env: module
Centaur.pascal.Module.Root: user
Centaur.pascal.Module.Location: centaur/tables/pascal/environment
```

El fichero de descripción del módulo, llamado `module.LM`, deberá contener las siguientes líneas:

```
defmodule centaur/tables/pascal/environment
files ("centaur/tables/pascal/environment/ICEditor.pascal.ll")
```

La inicialización del entorno

Para que el *ctview* adquiera el aspecto deseado al cargar un programa en pascal, se deben definir las dos funciones siguientes:

- {`pascal`}:`set-environment`, que toma como único argumento una instancia de {`ctview`}. Su misión es la de instalar el entorno adecuado a los programas Pascal.
- {`pascal`}:`clear-environment`, que también toma como argumento una instancia de {`ctview`}. Su misión es la destruir el entorno creado mediante la función anterior.

En la figura 8.8 se muestra el entorno utilizado para la edición del programa `newton-2.pascal`.

Asimismo, en la figura 8.9 se muestran los mensajes que aparecen en el `xterm` desde el cual se lanzó CENTAUR cuando se cargan los entornos para Pascal y para METAL.

La construcción del entorno

Como se observa en la figura 8.8, el entorno para Pascal está constituido por los siguientes elementos:

Figura 8.8: ICEeditor versión CENTAUR.

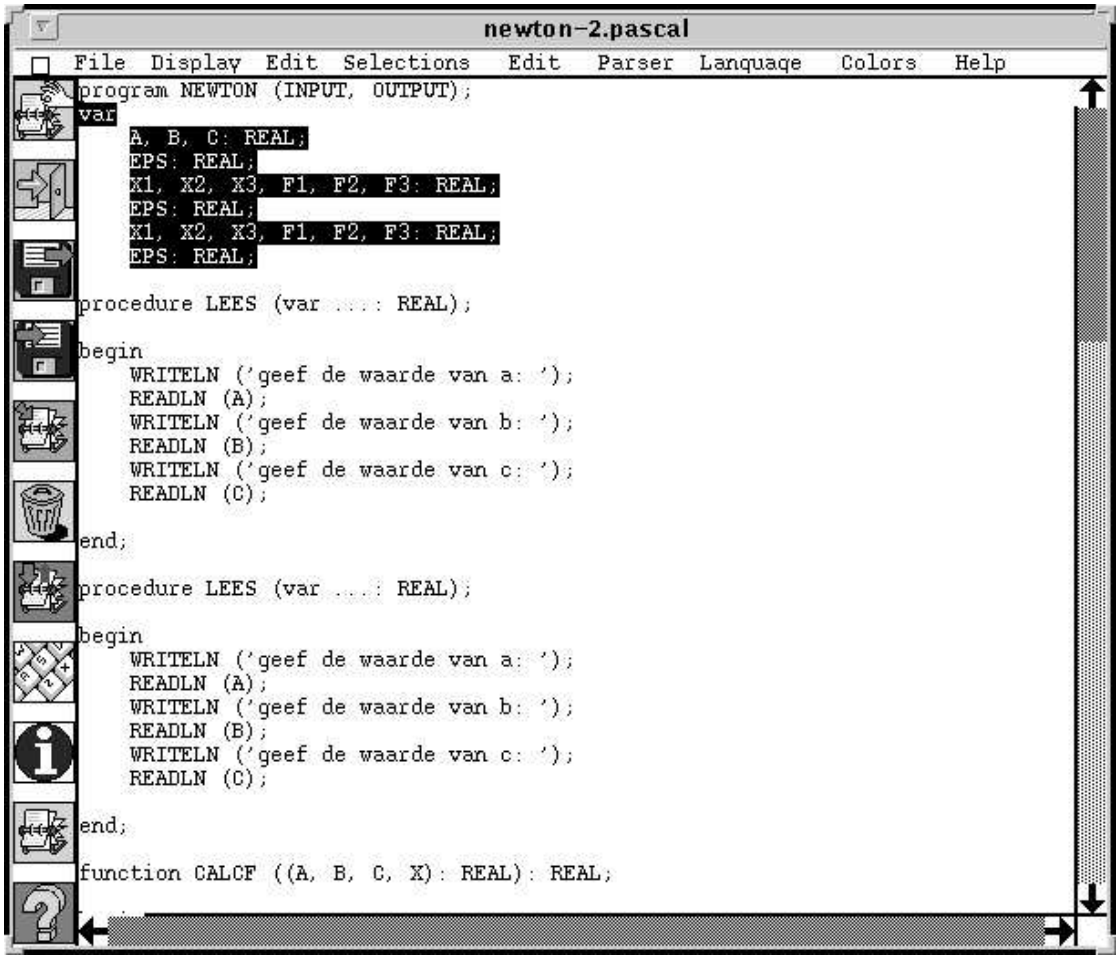
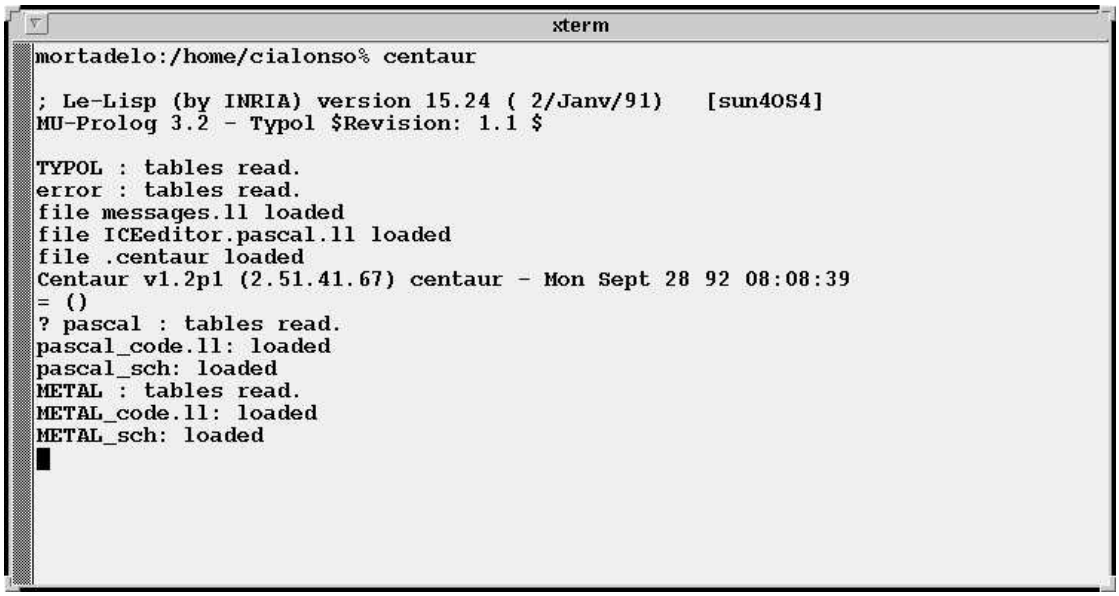


Figura 8.9: Mensajes adicionales en el `xterm`.



- Una barra de menú que contiene, además de las opciones habituales en los *ctview*'s, un conjunto de opciones adicionales específicas de ICEeditor.
- Una barra de botones en la cual se han situado los botones de ICEeditor.
- Un editor, limitado por sendas barras de desplazamiento vertical y horizontal.

El sistema gráfico de CENTAUR no permite mayores lujos a la hora de construir la interfaz gráfica. Con lo que está disponible se ha querido asimilar, en la medida de lo posible, la versión de ICEeditor realizada en AÏDA a las posibilidades gráficas de CENTAUR. A continuación se describe cada uno de los elementos enumerados anteriormente.

La barra de menú

Un *ctview* incorpora de modo estándar a la barra de menú los siguientes submenús:

- *File*, que permite leer y escribir programas en el *ctedit*.
- *Display*, que permite modificar ciertos parámetros del pretty printer, como son el nivel de impresión[9] y la anchura de la página, cambiar de pretty printer y redibujar el contenido del *ctedit*.
- *Edit*, que permite realizar operaciones genéricas de edición, como son cortar, copiar y pegar, y enviar expresiones previamente seleccionadas a un editor GNU Emacs independiente. En la figura 8.10 se muestra la ventana de un editor Emacs que contiene la parte del programa Pascal previamente seleccionada en la ventana que aparece en la figura 8.8. Para conseguir esta interacción entre Emacs y los editores de CENTAUR, es preciso incluir la siguiente línea en el fichero `.emacs.el` localizado en el directorio home del usuario:

  ```
  (load "/usr/local/centaur/etc/centaur.el")
  ```

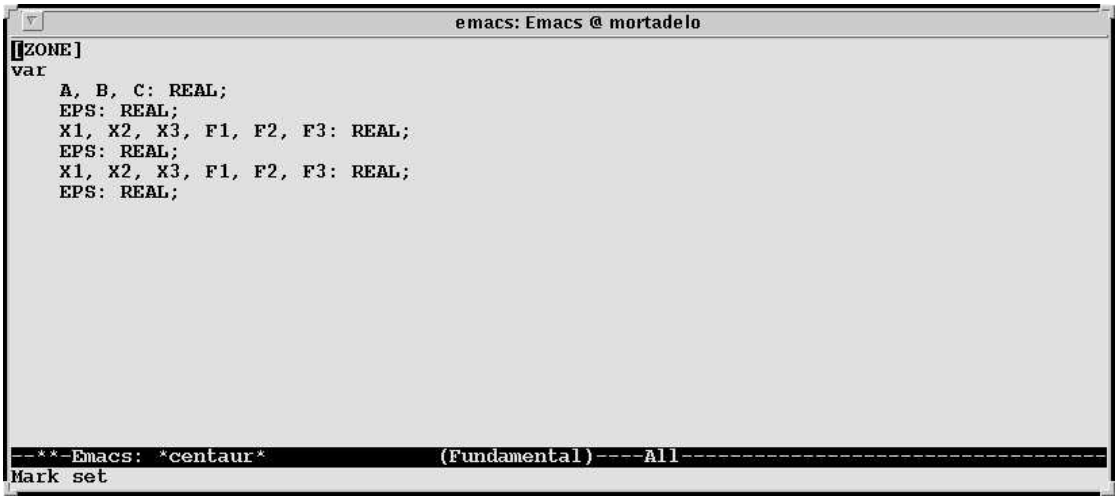
- *Selections*, que permite manipular en cierta medida las áreas seleccionadas: se las puede colapsar, expandir o deseleccionar.

Adicionalemte, se incorporan un conjunto de opciones de menú específicas de ICEeditor. Para ello se define la función `{pascal}:create-menu`, que toma como argumento un *ctview*, y modifica el menú estándar de éste añadiendo las siguientes opciones:

- *Edit*, que permite realizar las operaciones de edición de tokens típicas de ICEeditor: insertar, eliminar, modificar y solicitar información de un token.
- *Parser*, que permite realizar la compilación total e incremental del contenido del editor utizando ICE.
- *Language*, permite elegir el idioma de interacción con el usuario. Esta opción se ha pasado al primer nivel debido a que CENTAUR no permite enlazar menús desplegables entre sí.

[9] El nivel del árbol hasta el que se aplican las reglas descritas en la especificación del pretty printer.

Figura 8.10: Editor Emacs enlazado con CENTAUR.



- *Color*, permite elegir el color de fondo y de primer plano de los tokens editados.
- *Help*, proporciona ayuda del editor y de ICEeditor en general.

Para enlazar estas opciones al menú de un *ctview*, se envía el mensaje `add-menu` a dicho *ctview*, utilizando como argumentos el submenú y la posición que éste ocupará en la barra de menú.

La barra de botones

La barra de botones se ha de situar de modo obligatorio junto al borde izquierdo de la ventana. Está formada por los doce botones ya conocidos de la versión AÏDA:

- Botones de parser:
  - *Parser.*
  - *Parser-all.*
- Botones de manejo de ficheros:
  - *Load.*
  - *Save.*
- Botones de edición de tokens:
  - *Insert.*
  - *Delete.*
  - *Modify.*
  - *No action.*
  - *Info.*
- Botones de salida y ayuda:

- *Quit.*
- *Help.*

La función {pascal}:create-button-bar, que recibe como argumento un *ctview*, es la encargada de crear la barra de botones. Antes de enviar el mensaje add-to-column al *ctview*, es preciso enviar el mensaje show a cada uno de los botones, ya que de lo contrario no serán visibles.

Las funciones {pascal}:create-menu y {pascal}:create-button-bar son invocadas desde {pascal}:set-environment.

8.4 Conclusiones

El entorno CENTAUR posee un conjunto de potentes herramientas para la construcción de compiladores. Sin embargo, exhibe muestras de una cierta falta de integración entre esas herramienta y el entorno gráfico. Por otra parte, el conjunto de objetos gráficos disponible es ciertamente reducido, lo que limita mucho su utilización para la construcción de entornos amigables.

Sin embargo, la mayor dificultad viene dada por la deficiente documentación que acompaña al sistema. Si bien en los tres volúmenes de que consta [Centaur 92a, Centaur 92b, Centaur 92c] se detallan los aspectos de la definición de gramáticas utilizando los diversos formalismos, no ocurre lo mismo con la parte gráfica. Numerosas funciones de gran utilidad, por ejemplo las utilizadas para cargar los ficheros de iconos, no aparecen ni siquiera mencionadas. Lo mismo puede decirse de las funciones que son activadas al pulsar las opciones estándar incorporadas en los *ctview*. Tampoco se documenta adecuadamente el acceso a los árboles de análisis desde un *ctview*.

Sin embargo, el principal escollo a la hora de construir una interfaz viable para ICE utilizando CENTAUR lo constituye la dificultad de incorporar la incrementalidad y el no determinismo presentes en ICE dentro del formalismo METAL, de tal modo que éste sea capaz de reconstruir incrementalmente el árbol de análisis para que pueda ser utilizado por un programa PPML.

Apéndice A

Descripción general de ICE

En este apéndice se realiza una descripción general del generador de analizadores sintácticos no deterministas e incrementales que se utiliza en esta tesina: ICE[1].

ICE, desarrollado por Vilares como parte de su tesis doctoral [Vilares 92], toma como entrada una clase general de gramáticas de contexto libre. La forma en que se describe dicha gramática es idéntica a la que se utiliza con YACC, lo que permite que cualquier gramática previamente escrita para YACC se pueda utilizar también con ICE sin necesidad de realizar cambios.

Para asegurar la eficiencia computacional, ICE hace uso del concepto de traductor a pila no determinista como modelo operacional para simular todas las posibles computaciones, con una complejidad cúbica tanto espacial como temporal en el peor caso, aunque es lineal para una clase amplia de gramáticas. Otra característica fundamental es que no interpreta directamente las gramáticas, sino que primero las compila. El problema de la compartición óptima de computaciones lo resuelve empleando técnicas de programación dinámica.

ICE representa el análisis por medio de la secuencia de reglas que se deben usar para realizar una reducción de izquierda a derecha de la sentencia de entrada hasta alcanzar el símbolo inicial. Esta representación le permite resolver el problema de la compartición óptima de la estructra sintáctica de salida. Además, debido a su carácter incremental, al modificar una parte de la cadena de entrada tan sólo se eliminan las partes del análisis que habían sido generadas por los elementos de la cadena que han cambiado. La parte del análisis que permanece es la que se corresponde con la parte de la entrada que permanece invariable.

Desde un punto de vista empírico, ICE se muestra superior a otros analizadores de contexto libre y comparable a los analizadores sintácticos deterministas estándar cuando la cadena de entrada no es ambigua.

A.1 El análisis no determinista de ICE

Como todo analizador sintáctico de contexto libre, la misión de ICE es analizar sentencias de un lenguaje $\mathcal{L}(\mathcal{G})$ generado por una gramática $\mathcal{G} = (N, \Sigma, P, S)$ donde N es el conjunto

[1] **I**ncremental **C**ontext-free **E**nvironment.

de símbolos no terminales, Σ es el conjunto de terminales, P son las reglas y S el el símbolo inicial.

El modelo operacional es el de un traductor a pila o PDT[2] en el cual a partir de una *configuración inicial* (estado inicial, pila vacía, cadena de entrada completa y cadena de salida vacía) se van aplicando transiciones hasta alcanzar una *configuración final* (estado final y/o pila vacía, nada por analizar en la cadena de entrada y cadena de salida completa) que representa el resultado de un proceso de computación con éxito. Una sentencia ambigua para la gramática $\mathcal{G}$ provocará la existencia de varios *caminos* (secuencias de transiciones) válidos desde la configuración inicial hasta la final. La configuración final puede no ser única ya que diferentes caminos probablemente generarán diferentes cadenas de salida y además puede haber más de un estado final.

En el análisis de sentencias ambiguas, la compartición de computaciones es un problema fundamental. Sin embargo, en un PDT tradicional el formalismo de transiciones depende del contenido total de la pila con lo cual se reducen las posibilidades de compartición. Es por ello precisamente por lo que dicha estructura recibe la denominación de S^T. Para resolver este problema ICE hace uso de de *estructuras dinámicas*[3], introducidas originalmente en [Villemonte de la Clergerie 90] para el desarrollo de compiladores lógicos eficientes y aplicadas al análisis de contexto libre en [Vilares 92].

ICE realiza una interpretación mediante programación dinámica de un PDT consistente en la exploración sistemática de un espacio de elementos que reciben el nombre de *items*. Este espacio de búsqueda es una representación condensada de todas las posibles computaciones del PDT. ICE garantiza que realmente se exploran todas las partes útiles del espacio de búsqueda (completitud) y que las partes redundantes o inútiles se ignoran todo lo que sea posible (admisibilidad). Se puede probar [Vilares 92, Vilares 93] que la representación de configuraciones mediante items es compatible con el formalismo de transiciones y que la corrección y completitud de computaciones con los items se verifican en relación a S^T.

En la práctica, sólo se suelen considerar las estructuras dinámicas S^1 y S^2. Ambas construyen items a partir de la noción de *modo*. Un modo es una tupla de 4 elementos (p, X, S_i^w, S_j^w) donde p es el estado actual en el PDT, X es el último símbolo (terminal o no terminal) reconocido en el estado p, S_i^w es un puntero, llamado *back-pointer*, a la posición i de la cadena de entrada w que contiene el primer *componente léxico* derivado del símbolo X, mientras que S_j^w es un puntero a la posición j en la cadena de entrada del componente léxico que está siendo analizado. S^1 se diferencia de S^2 en que considera los items como estructuras formadas por un único modo , mientras que para S^2 los items están formados por dos modos de la forma $[(p, X, S_i^w, S_j^w)(q, Y, S_k^w, S_i^w)]$. Esto quiere decir básicamente que S^1 utiliza el tope de la pila para representar la configuración actual, mientras que S^2 utiliza el tope y su predecesor en la pila.

A fin de garantizar la eficiencia en un contexto no determinista, es preciso asegurar una buena compartición de computaciones. Es por ello que ICE hace uso de S^1, ya que S^2 no se puede considerar que sea óptima debido a la continua dependencia del contexto representado por el segundo modo de los items.

[2] **P**ush **D**own **T**ranslator.
[3] Dinamic frames.

A.1.1 El núcleo de ICE

Dada una gramática de contexto libre $\mathcal{G}$, mediante el uso de técnicas estándar se puede producir un *reconocedor* LALR(1), posiblemente no determinista, del lenguaje $\mathcal{L}(\mathcal{G})$.

Dada una cadena de entrada de longitud n, ICE asocia a cada componente léxico w_i, donde i representa su posición en dicha cadena, un conjunto de items llamado *itemset*.

Se generan nuevos itemsets mediante la aplicación de transiciones sobre los existentes, hasta que no es posible aplicar más transiciones. Un item representa la configuración de la pila en un momento dado: $[p, X, S_i^w, S_j^w]$, donde p es el estado en que se encuentra el autómata LALR(1) en dicho momento, X es el último símbolo gramatical reconocido en el estado p, S_i^w es el puntero de retroceso e indica el itemset que contiene el primer componente léxico derivado del símbolo X y S_j^w es el itemset que se está utilizando en ese momento.

Una transición $\tau = \delta(p, X, a) \ni (q, Y, u)$ en S^T, se traduce en S^1 de la siguiente forma:

1. $\tilde{\delta}([p, X, S_j^w, S_i^w], a) \ni ([q, \epsilon, S_i^w, S_i^w]), \epsilon)$
2. $\tilde{\delta}([p, X, S_j^w, S_i^w], a) \ni (I_0, I_0 \rightarrow a)$
3. $\tilde{\delta}([p, X, S_j^w, S_i^w], a) \ni (I_1, I_1 \rightarrow I_2)$
4. $\tilde{\delta}([p, \epsilon, S_j^w, S_i^w], a) \ni \tilde{\tau}_d = \tilde{\delta}_d([q, \epsilon, S_l^w, S_i^w], a) \ni ([q, \epsilon, S_l^w, S_j^w], I_3 \rightarrow I_4 I_5)$

si y sólo si:

1. $Y = X$
2. $Y = a$
3. $Y \in N$
4. $Y = \epsilon, \forall q \in Q$ tal que $\exists \delta(q, X, \epsilon) \ni (p, X, \epsilon)$

donde:

$$I_0 = [p, Y, S_i^w, S_{i+1}^w], \quad I_1 = [p, Y, S_i^w, S_i^w], \quad I_2 = [p, X, S_j^w, S_i^w]$$
$$I_3 = [q, \epsilon, S_l^w, S_i^w], \quad I_4 = [q, X, S_l^w, S_j^w], \quad I_5 = [p, \epsilon, S_j^w, S_i^w]$$

y:

$$\begin{cases} \tilde{\delta} & : \ \mathrm{It} \times \Sigma \cup \{\epsilon\} \longrightarrow \{\mathrm{It} \cup \tilde{\delta}_d\} \times \Pi^* \\ \tilde{\delta}_d & : \ \mathrm{It} \times \Sigma \cup \{\epsilon\} \longrightarrow \mathrm{It} \end{cases}$$

donde It el el conjunto de los items involucrados en el proceso de análisis y $\tilde{\delta}_d$ es el conjunto de *transiciones dinámicas*. Los casos precedentes se corresponden con:

1. Una acción *ir_a* del estado p al estado q bajo una transición X en el autómata LALR(1).
2. Meter en la pila el terminal a en el estado p. El nuevo item generado pertenecerá al siguiente itemset S_{i+1}^w.
3. Meter en la pila el no terminal Y en el estado p.

4. Sacar el tope de la pila en el estado p, donde q es un ancestro del estado p bajo la transición X en el autómata LALR(1). En este caso no se genera un nuevo item, sino una *transicion dinámica* $\tilde{\tau}_d$ con el fin de tratar la ausencia de información sobre el contenido del resto de la pila. Esta transición se puede aplicar no sólo a la configuración resultante de la primera, sino también sobre aquellas que sean generadas y que compartan la misma estructura sintáctica.

Ciertamente, la definición de las transiciones puede parecer confusa en un primer momento. Su significado adquiere una mayor claridad si se conoce el algoritmo de Earley de análisis sintáctico no determinista [Earley 70, Vilares 92], el cual, a partir de un itemset inicial, va aplicando a los items de cada itemset tres operaciones (*scanner*, *completer* y *predictor*) con las que se generan nuevos items y nuevos itemsets. El proceso finaliza cuando ya no se puede aplicar ninguna operación sobre ningún item. En Earley cada item tiene asociado una producción, un punto en esa producción que indica qué símbolos de la parte derecha han sido reconocidos hasta el momento y un puntero de retroceso al itemset de la posición en la entrada en la que se comenzó a buscar por la instancia de la producción asociada al item. La operación *scanner* consiste básicamente en leer el siguiente componente léxico de entrada y construir el correspondiente itemset. La operación *predictor* va aplicando las distintas reglas cuyo lado izquierdo es X cuando en un item el siguiente símbolo a reconocer es X. *Completer* es la operación que se aplica a un item cuando el punto está al final de la parte derecha de la producción y consiste en copiar del itemset indicado por el puntero de retroceso todos los items que tengan inmediatamente a la derecha del punto el no terminal de la parte izquierda de la producción, corriendo el punto una posición a la derecha, con lo que se indica que hay por lo menos un camino válido para reconocer ese símbolo para la parte de la entrada que se ha analizado.

ICE se diferencia de Earley en que tiene detrás un autómata, lo que le confiere una mayor eficiencia. Es también por ello que los items de ICE incluyen el estado del reconocedor LALR(1).

Lo que hace ICE es usar el autómata LALR(1) como base para las transiciones entre items. Básicamente, consiste en utilizar el autómata como una guía de modo que no se pierda tiempo realizando transiciones entre items que no estén el camino que lleve a alguna de las soluciones.

A.1.2 Un ejemplo sencillo

Para aclarar en lo posible el funcionamiento de los analizadores sintácticos generados por ICE, usaremos un ejemplo tomado de [Vilares 93].

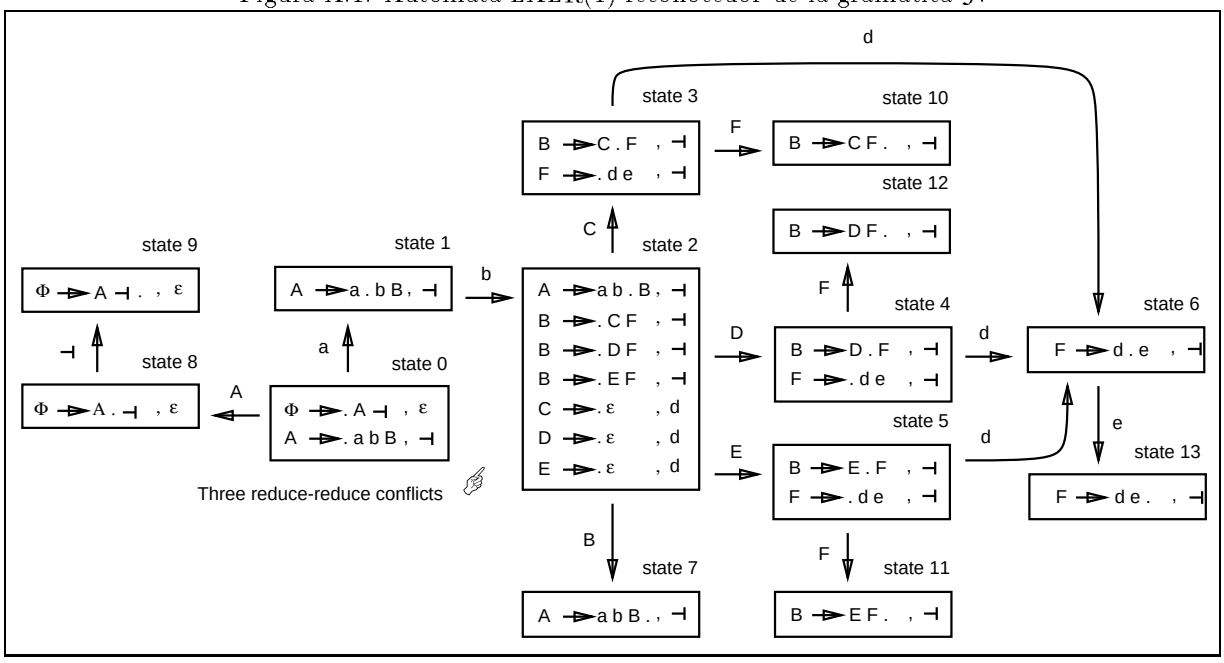
Se trata de analizar la entrada $w = abde \dashv$, según la siguiente gramática $\mathcal{G}$ en la que Σ es el símbolo inicial, las letras minúsculas son los componentes léxicos y las letras mayúsculas los no terminales:

$$
\begin{array}{lcll}
R0: & \Phi & \rightarrow & A \dashv \\
R1: & A & \rightarrow & a\ b\ B \\
R2: & B & \rightarrow & C\ F \\
R3: & B & \rightarrow & D\ F \\
R4: & B & \rightarrow & E\ F \\
R5: & C & \rightarrow & \epsilon \\
R6: & D & \rightarrow & \epsilon \\
R7: & E & \rightarrow & \epsilon \\
R8: & F & \rightarrow & d\ e
\end{array}
$$

En la figura A.1 se muestra el autómata LALR(1) reconocedor de la gramática $\mathcal{G}$. En [Aho et al. 90] se encuentra una buena descripción de cómo construir reconocedores de este tipo. Como la gramática es ambigua, en el estado 2 se producen 3 conflictos reducción-reducción. Esto se traduce en el ejemplo en tres derivaciones válidas para reconocer la entrada $abdc$, que son las siguientes:

$$
\begin{array}{llllllll}
\Phi & \Rightarrow & A \dashv & \Rightarrow abB & \Rightarrow & abCF & \Rightarrow & abF \Rightarrow abde \\
\Phi & \Rightarrow & A \dashv & \Rightarrow abB & \Rightarrow & abDF & \Rightarrow & abF \Rightarrow abde \\
\Phi & \Rightarrow & A \dashv & \Rightarrow abB & \Rightarrow & abEF & \Rightarrow & abF \Rightarrow abde
\end{array}
$$

Figura A.1: Autómata LALR(1) reconocedor de la gramática $\mathcal{G}$.



Al comenzar, el algoritmo añade automáticamente el item $I_0^0 = [0, \epsilon, S_0^w, S_0^w]$ (en Earley sería el item asociado a $[\Phi =\!\!\rightarrow .A \dashv]$). En la tabla A.1 se muestran los items involucrados en el proceso de análisis de nuestra entrada. En este momento estamos en el estado inicial 0 del autómata LALR(1).

Del item I_0^0 pasamos al item $I_1^0 = [0, \epsilon, S_0^w, S_0^w]$[4] aplicando una operación predictor.

[4] En Earley asociado a $A \rightarrow .abB$.

Ya que el punto está justo antes de la a y el siguiente componente léxico de entrada es precisamente a, aplicamos el operador scanner según Earley, con lo que obtenemos $I_0^1 = [0, a, S_0^w, S_1^w]$, el primer item del itemset S_1^w correspondiente al componente léxico en la posición 1[5]. Este item en Earley estaría asociado a $A \rightarrow a.bB$; a su vez, la máquina LALR(1) indica una transición al estado 1 con a. Por tanto, el estado actual es el estado 1.

Como el punto está justo antes de b y el siguiente componente léxico en la entrada es b, aplicamos la operación scanner con lo que obtenemos $I_0^2 = [1, b, S_1^w, S_2^w]$, el primer item del itemset S_2^w correspondiente al componente léxico en la posición 2[6]. Como el reconocedor LALR(1) indica una transición al estado 2 con b, el estado actual pasa a ser el estado 2.

En este estado se produce un triple conflicto de reducción, ya que se puede reducir indistintamente por las reglas $R2$, $R3$ y $R4$ para reconocer la cadena de entrada. YACC, que implemeta un reconocedor determinista LALR(1), resolvería este conflicto reduciendo por la primera regla e ignorando el resto. Sin embargo ICE, debido a su carácter no determinista, es capaz de tratar con ambigüedades como ésta y crear una estructura que representa los tres posibles árboles de análisis sintáctico con la mayor compartición posible[7].

Tabla A.1: Desarrollo para la entrada $w = abde\dashv$ en la gramática $\mathcal{G}$.

S_0^w		$I_0^0 = [0, \epsilon, S_0^w, S_0^w]$	S_4^w	$(w_4 = e)$	$I_0^4 = [6, e, S_3^w, S_4^w]$
S_1^w	$(w_1 = a)$	$I_0^1 = [0, a, S_0^w, S_1^w]$			$I_1^4 = [3, F, S_2^w, S_4^w]$
S_2^w	$(w_2 = b)$	$I_0^2 = [1, b, S_1^w, S_2^w]$			$I_2^4 = [4, F, S_2^w, S_4^w]$
		$I_1^2 = [2, C, S_2^w, S_2^w]$			$I_3^4 = [5, F, S_2^w, S_4^w]$
		$I_2^2 = [2, D, S_2^w, S_2^w]$			$I_4^4 = [2, B, S_2^w, S_4^w]$
		$I_3^2 = [2, E, S_2^w, S_2^w]$			$I_5^4 = [0, A, S_0^w, S_4^w]$
		$I_4^2 = [2, b, s_1^w, S_2^w]$	S_5^w	$(w_5 = \dashv)$	$I_0^5 = [8, \dashv, S_4^w, S_5^w]$
		$I_5^2 = [2, b, s_1^w, S_2^w]$			$I_1^5 = [9, \Phi, S_0^w, S_5^w]$
S_3^w	$(w_3 = d)$	$I_0^3 = [3, d, S_2^w, S_3^w]$			
		$I_1^3 = [4, d, S_2^w, S_3^w]$			
		$I_2^3 = [5, d, S_2^w, S_3^w]$			

En el estado 2, aplicando el operador predictor con la regla $R2$, obtendríamos $I_4^2 = [2, b, S_1^w, S_2^w]$, en Earley asociado a $B \rightarrow .CF$). El segundo y tercer elemento de I_4^2 se deben a que hasta el momento en este estado tan sólo se ha reconocido la entrada hasta b[8] y a que el itemset que contiene el primer componente léxico derivado de b es S_1^w[9].

El proceso que se seguiría a partir de aquí en caso elegir las reglas $R3$ o $R4$ para aplicar con predictor es análogo al que se va a mostrar a continuación. Es por ello que no se van a explicar esos dos caminos alternativos, ya que se alargaría demasiado el ejemplo y realmente no aportan nada nuevo a la comprensión del proceso.

En la situación actual, aplicamos el operador predictor con la regla $R5$ para volver a obtener $I_4^2 = [2, b, S_1^w, S_2^w]$.

[5] La primera letra en la cadena de entrada del ejemplo.

[6] Este item en Earley estaría asociado a $A \rightarrow ab.B$.

[7] Esto es, evitando triplicar las partes comunes a los distintos árboles.

[8] Un operador predictor no implica el reconocimiento de ningún símbolo gramatical: los terminales se reconocen con scanner y los no terminales con completer.

[9] Al ser b terminal, realmente no se puede derivar a nada, por lo que el puntero de retroceso debe ser forzosamente el itemset en el cual se reconoció el propio b.

Ahora podemos aplicar el operador completer para obtener $I_1^2 = [2, C, S_2^w, S_2^w]$. Con ello hemos reconocido el no terminal C en el itemset S_2^w[10]. Como la máquina LALR(1) indica que con C hay una transición al estado 3, éste pasa a ser el estado actual.

Al aplicar el operador predictor obtenemos $I_5^2 = [3, C, S_2^w, S_2^w]$ que estaría asociado en Earley a $F \rightarrow .de$; como el punto está justo delante de la d y el siguiente componente léxico en la entrada es precisamente d, aplicamos la operación scanner con lo que obtenemos $I_0^3 = [3, d, S_2^w, S_3^w]$ que es el primer item del itemset S_3^w[11]. El estado actual pasa a ser el 6 ya que así lo indica una transición el reconocedor LALR.

En el estado 6 aplicamos otra vez scanner para reconocer e y obtener el item $I_0^4 = [4, e, S_3^w, s_4^w]$ asociado según el algoritmo de Earley a $F \rightarrow de.$; el autómata LALR, por su parte, indica una transición al estado 13.

Ahora, según el algoritmo de Earley deberíamos realizar una operación completer. Para ello se utiliza el puntero de retroceso . Sin embargo, el significado del puntero de retroceso en ICE es diferente al que tiene en Earley. ICE, a diferencia de Earley, utiliza las transiciones de tipo 4 para regresar al estado donde se aplicó el predictor de F (el estado 3) y genera un item que indica el reconocmiento de F en ese estado con el mismo puntero de retroceso que cuando se hizo el predictor. El item obtenido tiene por tanto la forma $I_1^4 = [3, F, S_2^w, s_4^w]$. Este item estaría asociado en Earley a $B \rightarrow CF.$, por lo que a su vez ahora se debe relizar la operación completer para reconocer B.

Como el predictor para B se hizo en el estado 2 con puntero de retroceso S_2^w, el item resultante es $I_4^4 = [2, b, s_2^w, s_4^w]$ que en Earley estaría asociado a $A \rightarrow abB$, con lo cual se debe realizar el completer para reconocer A.

Debido a que la operación predictor para A tuvo lugar en el estado 0 y que el puntero de retroceso utilizado en ella había sido S_0^w, se obtiene el item $I_5^4 = [0, A, S_0^w, S_4^w]$ que en Earley estaría asociado a $\Phi \rightarrow A. \dashv$ con lo que estamos en condiciones de realizar un scanner sobre el delimitador final de la cadena[12]. Por su parte, el reconocedor LALR(1) indica una transición al estado 8.

Al realizar el scanner de $\dashv$ obtenemos el item $I_0^5 = [8, \dashv, S_4^w, S_5^w]$. La regla asociada según Earley sería $\Phi \rightarrow A \dashv .$, que indica el reconocimiento de toda la cadena de entrada. Efectivamente, el autómata indica que con $\dashv$ se debe realizar una transición al estado 9, que es un estado de aceptación, con lo cual finaliza el proceso.

La descripción que se ha realizado aquí del ejemplo es diferente a la contenida en [Vilares 93] ya que se ha tratado sobre todo de facilitar la comprensión rehuyendo de los estrictos formalismos de las transiciones en S^1, aunque para conseguirlo se haya tenido que tomar una más que discutible asimilación del comportamiento de ICE al del algoritmo Earley con el añadido de un autómata LALR(1). La descripción completamente formal puede encontrarse en [Vilares 93].

A.1.3 El bosque compartido

Como ya se comentó anteriormente, ICE representa el análisis por medio de las reglas de la gramática de contexto libre usadas en una reducción a derechas de la sentencia

[10] En Earley el item generado estaría asociado a $B \rightarrow C.F.$

[11] Estaría asociado en Earley a la producción $B \rightarrow d.e.$

[12] Este delimitador $\dashv$ no forma parte realmente de la cadena de entrada, sino que se añade para saber cuando se ha terminado de reconocer totalmente la cadena, hecho que sucede cuando se puede reducir al símbolo inicial Φ teniendo $\dashv$ como componente léxico a analizar.

analizada, en vez de utilizar directamente un árbol de análisis sintáctico. De este modo se consigue que cuando la sentencia tenga diferentes análisis, el conjunto de todas las cadenas de análisis sean representadas de forma finita compartida mediante una gramática de contexto libre que genera el conjunto completo, posiblemente infinito.

El grafo de bosque compartido es equivalente a un grafo AND-OR de una gramática de contexto libre, en donde las ramas AND se corresponden con árboles de análisis deterministas tradicionales y las ramas OR se corresponden con los distintos *caminos* creados por las ambigüedades en el análisis no determinista. La compartición de estructuras se logra permitiendo que un nodo tenga más de un padre. Se puede compartir un subárbol entero o una parte de los descendientes de un nodo dado.

Para generar el bosque compartido, los items realizan el papel de no terminales en una gramática de salida $\mathcal{G}_o = (N_o, \Sigma_o, P_o, S_o)$ donde N_o es el conjunto de todos los items, Σ_o el conjunto de los componentes léxicos de la gramática original $\mathcal{G}$ y P_o son las reglas construidas por el algoritmo de análisis junto con los items. Se genera una regla en la gramática de salida cada vez que se realiza una reducción o un desplazamiento en la pila. En ambos casos, el lado izquierdo de la regla se corresponde con el nuevo item que describe la configuración resultante. El lado derecha está formado, en el caso de un desplazamiento, por el componente léxico reconocido y en el caso de una reducción por los items que representan a los items eliminado de la pila. El último item producido por un proceso de computación que reconoce la cadena de entrada es el símbolo inicial S_o.

A.2 El análisis incremental en ICE

El análisis incremental de contexto libre de ICE recupera las partes del bosque compartido que permanecen estables entre dos pasos consecutivos de análisis, sin que ello sea costoso ni en espacio ni en tiempo.

Supongamos que $w_{1\ldots n}$ es la cadena original de entrada ya analizada. En el siguiente paso de análisis se tratará con la cadena de entrada modificada $x_{1\ldots n+k}$, con $k \in [-n, \infty)$.

Se denominan *items estables* a aquellos que representan una configuración estable en el PDT y que serían reconstruidos si se realizase un reanálisis completo de toda la entrada. Un item $I_i^w = [p, X, S_j^w, s_i^w]$ es estable si y sólo si existe un item $I_{w_i}^x = [p, X, S_j^w, S_{w_i}^x]$, es decir, un item es estable si es equivalente a un item del análisis de la nueva cadena. Para representar el concepto descrito, se utilizará la notación $I_i^w \equiv I_{w_i}^x$.

Desde el punto de vista práctico, ICE sólo considera el caso de la recuperación de itemsets completos, no de items sueltos. Esto es equivalente a recuperar todos los descendientes en las ramas OR de un nodo del bosque compartido. Aunque con ello no se evita la eliminación de todas las computaciones superfluas, sí que permite reducir notablemente el proceso de comparación entre las configuraciones de las pilas correspondientes a la cadena de entrada original y a la modificada.

A.2.1 Descripción del problema

Consideremos el caso de una única modificación en la cadena de entrada. Sea $w_{1\ldots n}$ la cadena original y $x_{1\ldots n+k}$, con $k \in [-n, \infty)$, la cadena modificada. La entrada x tendrá la

forma $x_{1...n+k} = w_1 \cdots w_l u_1 \cdots u_j w_{l+h+1} \cdots w_n$ con

$$\begin{cases} u_{1...j} \in \Sigma^* \\ h = \mid u \mid -k \end{cases}$$

donde

$$\mid u \mid = \begin{cases} k, \text{ si } x \geq 0 \\ 0, \text{ en cualquier otro caso} \end{cases}$$

Por tanto, la modificación ha consistido en sustituir $w_{l+1...l+h}$ por $u_{1...k}$. Concretamente, en el caso de una inserción tenemos que $\mid u \mid = k$ con $k > 0$, en el caso del borrado $u = \epsilon$ y $k < 0$ y en el caso de una sustitución $k = 0$.

A S_l^w se le denomina *punto de modificación relativa a w y x*. Según esto podemos distinguir tres clases de items según su participación en el nuevo proceso de análisis:

- Los items en $S_{1...l-1}^w$ se corresponden con la parte estable del proceso de análisis.
- Los items en $S_l^w S_1^u \cdots S_j^u$ se corresponden con la parte nueva del análisis.
- Los items en $S_{l+h+1...n}^w$ se corresponden con la parte del análisis que probablemente será recuperada.

A.2.2 La recuperación incremental

Se pueden considerar dos casos diferentes:

- *Recuperación total.* Se basa en detectar cuando el proceso de análisis se vuelve independiente de las modificaciones.
- *Recuperación parcial.* Se basa en aplicar la recuperación a todos los subárboles del bosque compartido correspondiente a una parte de la entrada que sigue a la parte modificada.

La recuperación total

Para que la recuperación total sea posible a partir de una posición i en la cadena de entrada es necesario asegurarse de que todas las futuras transiciones que eliminen elementos de la pila no dependan de la parte de la entrada modificada. Esto quiere decir que las modificaciones sólo deben causar perturbaciones *locales*, es decir, que sólo afectan a ramas interiores del bosque de análisis.

Para que se pueda realizar una recuperación completa a partir del itemset S_i^w correspondiente a la posición i de la entrada es condición suficiente que para cada item $I \in S_{i-1}^w$ que cumpla las siguientes condiciones, I sea estable y $t < l$, siendo l el punto de modificación relativa de w y x. Las condiciones son:

1. I es el argumento en una transición que elimina elementos de la pila realizada desde un punto en el sufijo de $w_{1...i}$. Es decir, desde la parte de la entrada que queda más a la derecha que w_i.
2. Todas las transiciones que eliminan elementos de la pila y que tienen a I como argumento, hacen la transición a un itemset S_t^w con $t < i - 1$.

A los items que satisfacen las condiciones 1 y 2 se les denomina $gaps^w_{w_i}$ o más simplemente $gaps^w_i$ cuando no hay confusión posible.

Una condición suficiente para la recuperación total es la siguiente: Dada una cadena $x_{1...n+k}$ con $k \in [-n, \infty)$, que tiene alguna modificación con respecto a $w_{1...n}$, y un punto S^w_l de modificación relativa de w y x, tal que se verifica que $\exists l \in [l+h+1, n)$ y

1. $gaps^w_l \sqsubset gaps^x_{w_1} (resp. gaps^w_l \equiv gaps^x_{wl})$
2. $\forall I \in gaps^x_{w_l}, I.back \leq l$

entonces, desde el punto de vista del reconocedor, $S^w_t \sqsubseteq S^w_{w_t}, \forall t \in [l, n) (resp. S^w_t \equiv S^x_{w_t}, \forall t \in [l, n))$

Esto quiere decir que si las transiciones que eliminan elementos de la pila son las mismas y la porción de texto que falta por analizar es la misma, entonces el proceso de análisis será idéntico a partir de ahí.

Este resultado se puede extender al caso de varias modificaciones simultáneas en la cadena de entrada definiendo un *punto recuperado totalmente de una modificacion relativa a w y x* como el punto en el proceso de recuperación tal que dicha modificación ha sido totalmente recuperada [Vilares y Dion 94].

Hasta aquí en lo que se refiere al reconocedor. En lo concerniente al analizador sintáctico, se debe encontrar además el alcance de las modificaciones en el bosque compartido inicial, una tarea relativamente *sencilla* ya que los nodos que se ven afectados por cambios en su estructura son los comunes con el nuevo bosque que tienen al menos un descendiente cambiado con respecto al nuevo desarrollo. Para actualizar uno de estos nodos es suficiente con encontrar sus descendientes estables en el bosque que han sido realmente recalculados y reemplazarlos por la estructura original correspondiente en el bosque de análisis recuperado.

Como sólo se recuperan itemsets completos, este fenómeno está limitado a los items que representan nodos triviales estables para los cuales sus ancestros en el bosque no se calculan en el mismo itemset S^w_l en el que ellos están incluidos. A esos items se les denomina $overgaps^w_{w_l}$ o simplemente $overgaps^w_l$ cuando no hay confusión posible.

Con el fin de extender la recuperación total al analizador, se debe asignar $I^w.forest := I^x.forest$ para todos los $I^w \in overgaps^w_l$, donde I^w y I^x representan el mismo nodo en el bosque de análisis, es decir $I^w \equiv S^x$.

La recuperación parcial

Si en la recuperación total se trataba de recuperar el bosque de análisis a partir de un un cierto itemset, en la recuperación parcial tan sólo se podrán recuperar intervalos sueltos de itemsets. Partiendo otra vez del caso de una modificación simple, diremos que es posible realizar una recuperación parcial del itemset S^w_i al itemset S^w_j cuando cualquier operación que elimina elementos de la pila no depende de las modificaciones realizadas entre $S^x_{w_i}$ y $S^x_{w_j}$ [13].

Por consiguiente, para que S^w_i permita la recuperación parcial a partir de él y hasta S^w_j es suficiente con que se cumpla que para cada item $I \in S^w_{i-1}$ que cumple las siguientes

[13] Nótese que los superíndices indican que estos últimos itemsets pertenecen a la cadena de entrada modificada y los primeros a a la original.

condiciones, no existe ninguna transición en $S^w_{i...j}$ que elimine elementos de la pila y que tome a I como argumento. Las condiciones son:

1. I es el argumento en una transición que elimina elementos de la pila desde un sufijo de $w_{1...i}$.
2. Todas las transiciones que eliminan elementos de la pila u que toman a I como argumento, retornan a un itemset S^w_t con $t < i-1$.

El cumplimiento de estas condiciones no implica, sin embargo, la estabilidad del item I, ya que no se tiene en cuenta el pasado del proceso de análisis que representa el back-pointer, como ocurría en el caso de la recuperación total. Formalmente, diremos que un item $I^w_i = [p, X, s^w_j, S^w_i]$ es *débilmente estable* si y sólo si existe un item $I^w_{w_i} = [p, X, S^x_{w_j}, S^x_{w_i}]$ y se denotará como $I^w_i \cong S^x_{w_i}$. Los items I^w_i y $S^x_{w_i}$ no representan necesariamente subárboles equivalentes en el bosque compartido. La relación de inclusión provocada por $\cong$ se denota $\prec$.

Si consideramos de nuevo la definición de $gaps^w_i$, podemos concluir que no hay transiciones que eliminen elementos de la pila en $S^w_{i...k}$ que tomen un $I \in gaps^w_i$ como argumento. Desde un punto de vista formal, se puede asegurar que dados:

- $x_{1...n+k}, k \in [-n, \infty)$, una cadena de entrada modificada a partir de $w_{1...n}$
- $S^w_{l_{1...m}}$, que representa m puntos contiguos de modificacion relativos a w yx,

que verifican $\exists i \in [1, m], l, j \in [l_i + h_i + 1, l_{i+1})$, tal que:

1. $gaps^w_l \preceq gaps^x_{w_l}$ $(resp. gaps^w_l \cong gaps^x_{w_l})$
2. S^w_j es el primer itemset que aplica una operación *pop* sobre $gaps^w_l$

entonces, desde el punto de vista del reconocedor, $S^w_t \sqsubseteq s^x_{w_t}, \forall t \in [l, j-1]$ $(resp. S^w_t \equiv S^x_{w_t}, \forall t \in [l, j-1])$. Para extender este resultado al bosque de análisis compartido, es suficiente con realizar la asignación $I^w.forest := I^x.forest$, donde $I^w \cong I^x$, para todos los $I^w \in overgaps^w_l$.

Apéndice B

Introducción a AÏDA

La implementación que se muestra en esta tesina ha sido realizada utilizando la herramienta AÏDA de ILOG [ILOG 92c]. Esta herramienta ha sido diseñada para facilitar la creación de interfaces gráficas en entornos de ventanas. Está construida sobre la base del LE-LISP [INRIA 91] e incluye una librería de objetos gráficos y un conjunto de herramientas especializadas en la manipulación de dichos objetos y en el desarrollo de aplicaciones.

La utilización de AÏDA presenta una serie de ventajas entre las cuales podemos citar las siguientes:

- **Portabilidad**. En efecto, AÏDA está disponible para la práctica totalidad de las máquinas que corren bajo el sistema operativo UNIX, para los entornos DEC-VAX bajo VMS y para plataformas Intel y Motorola.

- **Rápido prototipado**. Al estar construida sobre la base de un potente lenguaje con orientación a objetos como es ceyx [INRIA 91], podemos disponer de las ventajas que proporciona este paradigma de programación, entre las cuales se encuentra la posibilidad de desarrollar rápidamente un prototipo que funcione mediante la combinación de objetos predefinidos, lo que nos permitirá evaluar pronto la idoneidad del diseño inicial.

- **Modularidad**. Esta es una característica derivada tanto de la orientación a objetos como de la naturaleza pseudo-funcional del LE-LISP, que permiten una descomposición del problema a resolver en subproblemas. Cada uno de estos subproblemas se podrá descomponer así mismo en subproblemas de nivel inferior y así sucesivamente. Las interacciones entre los subproblemas deben realizarse mediante interfaces bien definidas.

- **Flexibilidad**. Como consecuencia de lo anterior, se puede modificar la implementación de un módulo sin que afecte a los restantes. Basta con mantener las interfaces. Asimismo, se pueden incrementar las funcionalidades de un módulo sin que ello afecte a los módulos ya existentes.

- **Eficiencia**. Las aplicaciones desarrolladas se caracterizan por su eficiencia. Esto se debe a la posibilidad de realizar una compilación de los programas, a la buena implementación del LE-LISP y a la posibilidad de enlazar con funciones escritas en

C, de modo que aquellos aspectos de una aplicación en los que se precise una muy elevada rapidez puedan ser adecuadamente resueltos.

Se ha tratado de conjugar todas estas características para conseguir una implementación modular fácilmente extensible. Las versiones concretas que se han utilizado son la 15.25 de LE-LISP, la 1.65 de AÏDA y la 1.35 de MASAÏ, ejecutándose en los siguientes entornos:

- Sistemas IBM RISC System/6000 bajo AIX 3.2.5 con AIX Windows 1.2.5 (X11R5).
- Sistemas Sun SPARC bajo Solaris 1.1, esto es, SunOS 4.1.3 con OpenWindows 3 (X11R5).

B.1 LE-LISP

Como AÏDA [ILOG 92c] está construido sobre LE-LISP [INRIA 91], es conveniente realizar primero una breve introducción sobre las características de este lenguaje antes de pasar a describir los elementos propios de la herramienta. Como su propio nombre da a entender, LE-LISP pertenece a la amplia familia de dialectos de LISP. Su nacimiento tuvo lugar en el INRIA[1] de Rocquencourt (Francia) a principios de los años 80. En aquellos momentos, en dicho instituto se estaba utilizando LISP como lenguaje de implementación de un complejo sistema de diseño VLSI que incluía gráficos, simulación, etc. Debido a la rápida evolución experimentada por el hardware en aquellas fechas, el proyecto VLSI se fue desarrollando progresivamente sobre muchas máquinas heterogéneas e incompatibles entre sí. La inexistencia de una variante de LISP que se pudiese ejecutar en todas las máquinas utilizadas suponía una seria complicación. Es por ello que se decidieron a crear una variante propia del LISP en la que la portabilidad, tanto del propio sistema LISP como de las aplicaciones, fuese una característica fundamental.

LE-LISP fue diseñado para obtener una máxima eficiencia tanto en términos de los recursos necesarios para su ejecución[2] como de los necesarios para el desarrollo de aplicaciones[3]. Para conseguir estas metas la implementación del sistema se realizó en C. Con el fin de obtener la portabilidad necesaria, se diseñó una *máquina virtual* denominada LLM3, sobre la cual se construyó el sistema LE-LISP, con lo cual tan sólo es necesario portar LLM3 de una máquina a otra para garantizar que el resto del sistema funcionará correctamente. Además, está disponible un lenguaje de la máquina virtual LLM3 denominado LAP[4] a partir del cual se puede tener acceso a los recursos internos de LLM3, lo cual abre la posibilidad de escribir nuevas funciones estándar o incluso de construir un nuevo compilador de LE-LISP o de otro lenguaje. Con el tiempo, el sistema LE-LISP ha ido creciendo al incorporar mejoras como son la aritmética genérica extensible por el usuario, un sistema de tipos orientado a objetos, bibliotecas gráficas virtuales, acceso a rutinas externas, etc.

[1] **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique.
[2] Memoria y tiempo de ejecución.
[3] Tiempos de desarrollo y configuración necesaria del ordenador utilizado.
[4] **L**isp **A**ssembly **P**rogram.

A continuación se realizará una descripción de LE-LISP que pueda servir de base para una mejor comprensión de las utilidades de AÏDA que se mostrarán en la sección B.2. Por tanto, lo que viene a continuación no pretende ser una guía de programación en LISP.

B.1.1 Los objetos LE-LISP

El lenguaje LE-LISP opera con objetos llamados *expresiones simbólicas* o más brevemente *S-expresiones*, que se pueden clasificar como sigue:

- Objetos atómicos.
  - Símbolos[5].
  - Números.
  - Cadenas de caracteres.
- Objetos compuestos.
  - Listas.
  - Vectores.

Símbolos

Son identificadores que nombran variables, funciones o etiquetas. LE-LISP representa un símbolo como un puntero a un descriptor localizado en una zona especial de memoria. Cada descriptor tiene las siguientes propiedades intrínsecas:

- **c-val** (*cell-value*): contiene el valor de un símbolo que es una variable. El acceso a este valor es extremadamente rápido. Cuando se crea un símbolo, su c-val es indefinido. Cualquier intento de referenciar un símbolo que no tiene asignado un valor provocará un error.
- **p-list** (*property-list*): contiene la lista de propiedades del símbolo en forma de P-lista.
- **f-val** (*function-value*): contiene el valor asociado a un símbolo que es considerado como una función. Dicho valor es una posición de memoria para funciones de tipo SUBR y una S-expresión si es de tipo EXPR, FEXPR, MACRO o DMACRO.
- **f-type** (*function-type*): contiene el tipo de función almacenado en f-val.
- **p-type** (*print-type*): contiene la información necesaria para editar la representación externa del símbolo.
- **o-val** (*object-value*): Se utiliza en la implementación de las extensiones orientadas o objeto. Puede contener cualquier S-expresión.
- **a-link** (*atom-link*): contiene la dirección del siguiente símbolo en la tabla de símbolos.

[5] Generalmente son nombres de variables o funciones.

- **pckgcell** (*package cell*): contiene el nombre del *paquete* [6]. Una de las principales utilidades de los paquetes es que proporcionan la base para una eficiente ejecución de los métodos en las extensiones de programación orientada al objeto.
- **p-name** (*print-name*): contiene la dirección de la cadena de caracteres que representa el nombre del símbolo.

No es aconsejable que el usuario manipule directamente las propiedades intrínsecas de los símbolos.

Números

LE-LISP utiliza enteros de 16 bits, mientras que los números en coma flotante se representan utilizando 31, 32 48 o 64 bits, dependiendo de la implementación concreta. También incorpora bibliotecas para la manipulación de números de precisión arbitraria.

Cadenas de caracteres

Se representan externamente encerrando entre comillas los caracteres que los componen. Si la cadena contiene en sí misma el carácter de las comillas, éste se consigue insertando dos comillas consecutivas en la posición adecuada de la cadena.

El tipo de una cadena de caracteres es `string` por defecto, aunque puede cambiarse a voluntad mediante el uso de la función `typestring`.

Listas

LE-LISP utiliza para representar las listas el formalismo estándar del CAR y del CDR. Brevemente, diremos que el primer elemento de una lista constituye su CAR, mientras que el resto de la lista constituye el CDR. La función `cons` se utiliza para construir listas, de modo que `(cons car cdr)` construye la lista que tiene como CAR a `car` y como CDR a `cdr`.

Las listas se representan encerrando sus elementos entre paréntesis, separando los distintos componentes, que pueden ser de tipos diferentes, mediante espacios[7]. La notación especial del *par punteado*, como por ejemplo `(1 . "pepe")`, se utiliza cuando el CDR es un átomo.

La lista vacía está identificada con el átomo NIL. Entre otras consecuencias, esto implica que NIL se puede considerar al mismo tiempo como:

1. Un átomo cualquiera llamado NIL.
2. La lista vacía.
3. El valor lógico Falso.

En relación al tercer caso, diremos que cualquier objeto LE-LISP distinto de NIL se considera, desde el punto de vista de su valor lógico, como verdadero. Si no se pone una

[6] Un paquete (*package*) tiene la forma $\#:<symbol_1>:<symbol_2>:\ldots:<symbol_n>$ y se utiliza para establecer una jerarquía de nombres.

[7] Por jemplo `(1 total "pepe" 4.2)` es una lista de cuatro elementos.

quote[8] delante de una lista, se entiende que su primer elemento es una función que toma como argumentos los siguientes elementos de la lista.

Vectores

Se corresponde con el tipo de los arrays unidimensionales que poseen la mayoría de los lenguajes de programación. Un vector es una colección de objetos LE-LISP a los cuales se puede acceder por medio de un índice, su posición en el vector, teniendo en cuenta que el primer elemento ocupa la posición 0. El tipo por defecto de un vector es `vector`, pero puede ser cambiado a voluntad de modo similar a como acurría en el caso de las cadenas de caracteres, pero utilizando en este caso la función `vector`.

Los vectores se representan de la forma $\#[S_1, S_2, \ldots, S_n]$, donde los S_i representan S-expresiones. El acceso a los elementos de un vector es muy rápido, aunque en las implementaciones de LE-LISP el número de elementos de un vector está limitado a 32767. Los vectores se almacenan en una zona especial de la memoria que se compacta dinamicamente mediante un algoritmo de *garbage collection*[9] que es lineal en tiempo.

Como el valor de un vector es el vector en sí mismo, no hay necesiad de utilizar la *quote* como en las listas. Aunque los vectores son unidimensionales, como un vector es a su vez una S-expresión, sí están permitidos los vectores de vectores.

B.1.2 Otros tipos de datos importantes

Arrays

LE-LISP implementa funciones para el manejo de arrays multidimensionales. `makearray` permite crear un array multidimensional, mientras que `aref` y `aset` permiten obtener y establecer, respectivamente, el valor asociado a una posición del array.

Tablas hash

Las tablas hash se incorporaron en la versión 15.2 de LE-LISP. Permiten procesar pares formados por una *clave* y un *valor* de una manera más eficiente que las listas de asociación. En las tablas hash el tiempo de búsqueda no es función del número de elementos contenidos en la tabla, sino de una constante que depende únicamente del estado de la tabla hash.

Las tablas hash son adaptativas, ya que minimizan la constante del tiempo de búsqueda y al mismo tiempo optimizan el uso del espacio de memoria reservado para el almacenamiento de las asociaciones clave/valor.

El valor de una tabla hash es la tabla hash en sí misma, por lo que no es necesario utilizar la *quote* con ellas.

Hay dos clases de tablas hash, diferenciándose en si la función de búsqueda de claves utiliza el predicado `equal` o el predicado `eq`. Las primeras se crean con `make-hash-table-equal` y las segundas con `make-hash-table-eq`.

Las siguientes funciones operan sobre tablas hash:

- `clrhash` borra todo el contenido de una tabla hash, aunque no elimina la tabla hash en sí.

[8] El carácter '.
[9] Recogida de basura.

- `hash-table-p` indica si un objeto es una tabla hash.
- `puthash` añade un nuevo par clave/valor.
- `get-hash` retorna el valor asociado a una clave.
- `remhash` borra una clave y su valor asociado de la tabla.
- `hash-table-count` devuelve el número de pares clave/valor almacenados.
- `maphash` permite aplicar una función a todos los pares de una tabla hash.

En la implementación descrita en esta tesina, se utilizan tablas hash para almacenar la correspondecia entre el texto y los tokens resultantes del análisis de la entrada.

Conjuntos matemáticos

LE-LISP incluye funciones que permiten manejar listas como si fuesen conjuntos. En una lista que va a ser utilizada como conjunto el orden de los elementos no es relevante. De hecho, cuando una de tales listas es el valor devuelto por una función, el orden en que estarán sus elementos es impredecible.

B.1.3 Tipos de funciones

La función básica que utiliza el intérprete LE-LISP para evaluar expresiones es `eval`, de modo que `(eval <expresión>)` evalúa `<expresión>`. Sin embargo, el modo en que se realiza esa evaluación reviste diferentes características según el tipo de objeto que se vaya a evaluar:

- En el caso de un símbolo, su evaluación consiste en la recuperación del campo **c-val**
- En el caso de números, cadenas de caracteres y vectores, el resultado de la evaluación es el propio objeto evaluado.
- En el caso de las listas, LE-LISP siempre considera que son llamadas a funciones. De hecho, las listas son referenciadas como *formas* donde el CAR es la función y el CDR la lista de argumentos. En el caso de las funciones anónimas, el CAR viene dado por una lista especial de la forma `(lambda (<parámetros>) (<cuerpo-de-la-función>))`.

En relación al tipo de evaluación aplicada, se pueden distinguir cuatro tipos de funciones:

- Funciones que evalúan sus argumentos (tipo **expr**).
- Funciones que no evalúan sus argumentos (tipo **fexpr**).
- Funciones macro sencillas (tipo **macro**).
- Funciones macro de sustitución (tipo **dmacro**).

A continuación vamos a analizar más detalladamente cada uno de los distintos tipos de funciones.

Funciones expr

Esta clase de funciones están escritas en el propio LE-LISP. Su característica fundamental es que los argumentos son siempre evaluados. Se definen generalmente utilizando la función `defun`. El proceso de evaluación se realiza de la siguiente manera:

1. Los valores de los nombres de los parámetros se guardan en la pila mientras se les asocian los nombres de los argumentos. Esto quiere decir que en este tipo de funciones las llamadas se realizan *por valor*.

2. Se evalúan las expresiones en el cuerpo de la función. El resultado devuelto es el valor resultante de la evaluación de la última de dichas expresiones.

3. Se deshacen las asignaciones realizadas en el primer paso. Se restauran los valores anteriores de los nombres de los parámetros, que habían sido guardados en la pila.

Como caso particular, en aquellas funciones expr cuya lista de parámetros contine únicamente `&nobind`, no se asigna ninguna variable. La función `arg` sin argumentos se utiliza en el cuerpo de dichas funciones para conocer el número de argumentos pasados en cada llamada, mientras que `(arg n)` devuelve el argumento de la posición `n`, considerando que el primero ocupa la posición 0. Esta clase particular de funciones se utiliza para crear funciones con un número variable de argumentos.

La compilación de funciones de tipo expr da lugar a funciones LLM3 que reciben el nombre de **subr**, de las cuales existen entre 400 y 500 en el sistema.

Funciones fepxr

Son funciones escritas en LE-LISP y evaluadas por las funciones de evaluación estándar `eval`, `apply` o `funcall`. Se definen por medio de la función `df`. Su característica fundamental es que no evalúan sus argumentos, sino que es el programador el encargado de evaluarlos mediante la utilización de `eval` en el cuerpo de la función.

La compilación de este tipo de funciones da lugar a las funciones LLM3 denominadas **fsubr**. Este tipo de funciones se usan principalmente como funciones de control o para la manipulación de nombres y reciben generalmente el nombre de *formas especiales*. Su número es reducido en LE-LISP.

Funciones macro

Para definirlas se utiliza la función `dm`. Tiene un número variable de argumentos, que no se evalúan. Para evaluar una forma que tiene a una macro como función, el evaluador primero evalúa la función asociada con esa macro utilizando la forma entera (obviamente, no evaluada) como argumento. Entonces re-evalúa el valor devuelto por esta primera evaluación. Por consiguiente, la evaluación de una macro es un proceso de dos pasos.

La compilación de este tipo de funciones da como resultado funciones **msubr** en LLM3.

Funciones dmacro

Se definen por medio de la función `defmacro`. La evaluación de este tipo de funciones difiere de la realizada en las de tipo macro en dos aspectos:

- Se utiliza como argumento el CDR, sin evaluar, de la forma.
- Después de realizar la primera evaluación, se reemplaza físicamente la forma entera por el valor retornado.

Son menos generales que las funciones macro, pero se usan con más frecuencia que aquellas. Su compilación da lugar a funciones de tipo **dmsubr** en LLM3.

B.1.4 Programación orientada a objetos con LE-LISP

LE-LISP permite definir *objetos estructurados* del tipo de los `record` en Pascal, los `struct` en C o los `structure` en Fortran. Para ello se usa la primitiva `defstruct`. Estos objetos estructurados están formados por un cierto número de *campos* no tipados. Se puede definir una estructura como sub-estructura de una precedente, lo cual permite crear una jerarquía de tipos con *herencia*. Sin embargo, no se soportan jerarquías múltiples, aunque ello no representa una limitación en la práctica, pues este tipo de herencia plantea numerosos problemas cuando se utiliza. A las estructuras se les pueden asociar *métodos*, con lo cual se puede conseguir *encapsulamiento* de funciones y datos, al estilo de los `object` del Pascal o de las `class` del C. También se pueden conseguir *polimorfismo* definiendo métodos con el mismo nombre en objetos distintos para realizar acciones similares. La primitiva `send` permite enviar un mensaje a un objeto (en el sentido de O.O.[10].). El sistema se ocupa automáticamente de determinar cual es el método adecuado que debe aplicar al objeto para satisfacer el mensaje.

Definición de estructuras

Al definir los objetos estructura, además de definir los campos que lo forman, se puede determinar un valor incial para ellos, que será asignado en ese campo en el momento de creación de cada nueva instancia.

Como ejemplo, a continuación definimos el tipo `person` como una estructura con los campos `name`, `DNI` y `birthday`. A este último campo se le asigna un valor inicial cada vez que se crea una instancia de `person`. Dicho valor es el resultado de la evaluación de la función `get-date` que supongamos devuelve la fecha actual.

```
(defstruct person
       name
       DNI
       (birthday (get-date)))
```

Creación de instancias

Para asignar a `Manolo` una instancia de `person`, podemos utilizar las dos opciones siguientes:

```
(setq Manolo (new 'person))
(setq Manolo (#:person:make))
```

Esto se debe a que cuando se crea una estructura, automáticamente se crea el método `make` (que devuelve una instancia de la estructura) y un método para acceder a cada campo cuyo nombre coincide con el del campo.

[10] Orientación a Objetos

Resumiendo, diremos que `defstruct` representa la parte declarativa de la programación con tipos estructurados, es decir define las *clases*, mientras que `make` y `new` generan *instancias* concretas de esas clases (que muchas veces reciben el nombre de *objetos* en el ámbito de la O.O.)

Métodos

Mediante la sintaxis `(#:estructura:campo objeto)` podemos recuperar el valor del campo `campo` en el objeto `objeto`, que debe ser una instancia de `estructura`. Si lo que queremos es asignar un valor a un campo, usaremos la sintaxis `(#:estructura:campo objeto valor)`.

Podemos definir un nuevo método llamado `age` que dado un objeto `person` devuelva como resultado su edad de la siguiente manera[11]:

```
(defun #:person:age (pers)
       (- (year (get-date)
          (year (#:person:age pers)))
```

En general, la forma en que se define un método coincide con la forma usual de definir funciones con la salvedad de que el primer parámetro estará ocupado por el objeto receptor del método y que se debe especificar la estructura a la pertenece el método mediante la sintaxis `#:estructura:método` o una forma abreviada equivalente.

Jerarquías

La utilización de `defstruct` permite la definición de jerarquías de tipos, es decir, un árbol de tipos en donde las estructuras se corresponden con los nodos y donde los descendientes de un nodo *heredan* las características de dicho nodo, esto es, sus campos y métodos. El nodo raíz del árbol se denota por #.

Como ejemplo, a continuación se define los tipos `lecturer` y `student` que son subtipos de `person`[12] y se crea una instacia para cada uno de ellos:

```
(defstruct #:person:lecturer
        category)

(defstruct #:person:student
        degree)

(setq Manolo (#:person:lecturer:make))
(setq Pablo (#:person:student:make))
```

Para obtener la edad de `Manolo` y `Pablo` podemos hacer:

```
(setq manolo-age (#:person:age Manolo))
(setq pablo-age  (#:person:age Pablo))
```

o bien:

```
(setq manolo-age (send 'age Manolo))
(setq pablo-age  (send 'age Pablo))
```

[11] Suponiendo que **`year`** y **`get-date`** son funciones ya definidas.
[12] Aunque en el mundo real a veces no lo parezca.

El uso de `send` es más ineficiente ya que el sistema LE-LISP debe recorrer ascendentemente la jerarquía hasta encontrar el método cuyo nombre coincide con el mensaje, aunque evita al programador el tener que preocuparse del lugar en la jerarquía en que están definidos los métodos. La función `send-super` es similar a `send` excepto que comienza a buscar por el método en el nodo padre del objeto pasado como argumento. Otras variantes son `send-error`, `csend` y `send2`.

Por otra parte, al ser posible redefinir los métodos a lo largo de la jerarquía, `send` aplica siempre el método más cercano ascendentemente al objeto de aplicación, mientras que indicando el nombre completo podemos especificar qué método exacto queremos ejecutar.

Abreviaturas

Para evitar tener que escribir la jerarquía completa como en `#:person:lecturer` existen las *abreviaturas*, que se definen usando `defabbrev`. Por ejemplo, si definimos `defabbrev profesor #:person:lecturer`, a partir de este momento podremos usar {`lecturer`} como sustituto de `#:person:lecturer` en cualquier lugar donde esto último era válido. Las funciones `put-abbrev`, `get-abbrev`, `abbrevp`, `has-an-abbrev` y `rem-abbrev` también están relacionadas con las abreviaturas.

B.1.5 Mesajes multi-idioma

Desde el punto de vista del usuario, un idioma es simplemente un símbolo. En LE-LISP hay dos lenguajes predefinidos, `english` y `french`. Cualquier mención a un idioma que no ha sido definido provocará un error.

Mediante `(record-language 'spanish)` podemos definir un nuevo idioma, en este caso el `spanish`.

Un mensaje es una variable a la que se asocian las distintas traducciones de un *mesaje* (en el sentido cotidiano de la palabra) en los idiomas registrados. Por ejemplo,

```
(defmessage #:ICEeditor:file-msg
        (english "File")
        (french  "Archive")
        (spanish "Archivo"))
```

permite que cuando tengamos que escribir la palabra "archivo", podamos usar `(get-message #:ICEeditor:file-mag)` para obtener la cadena de caracteres asociada a dicha palabra en el idioma utilizado en ese momento. Las otras funciones utilizadas con el manejo de los mensajes multi-idioma son:

- `remove-message`, que permite eliminar un mensaje de la base de datos de mensajes.
- `put-message`, que permite asignar una cadena de caracteres a un mensaje en un idioma particular.
- `current-language` permite establecer el idioma que se va a usar en adelante.
- `default-language` permite definir qué idioma se usará en aquellos mensajes que no tiene un string definido para el idioma usado en un momento dado.
- `remove-language` permite eliminar un idioma y todos sus mensajes asociados de la base de datos de idiomas.

- `message-languages` devuelve la lista formada por todos los idiomas registrados en un momento dado.
- `get-all-messages` devuelve la lista de todos los mensajes definidos en un idioma concreto, por lo que el programa

  ```
  (mapcar (lambda (x) (cons x . (get-message x))) (get-all-messages 'english))
  ```

  devuelve una lista de pares con los mensajes y su string asociado en inglés. Esto es útil para posteriormente traducir cada mensaje e incorporarlo a la base de datos de mensajes en el nuevo idioma mediante la aplicación de `put-message` a cada par de la lista traducida.

B.1.6 Virtual bitmap display

El sistema LE-LISP proporciona una utilidad de gráficos denominada *Virtual Bitmap Display* o abreviadamente VBD con la que consigue gestionar una pantalla de mapa de puntos y un dispositivo apuntador de modo independiente al hardware subyacente. La portabilidad del VBD ha permitido su migración a diferentes sistemas de ventanas, servidores y gestores.

Pantallas

Una pantalla (*screen*) es una estructura LE-LISP que permite manipular el dispositivo físico de la pantalla. Las funciones del VBD permiten construir simultáneamente pantallas sobre diferentes dispositivos. Esto es útil, por ejemplo, cuando queremos mostar ventanas en más de un terminal gráfico. La función `bitprologue` crea, inicializa y devuelve como resultado una pantalla. Con `current-display` podemos hacer que una de las pantallas previamente creadas se convierta en la actual. Todas las funciones del VBD actúan sobre la pantalla actual excepto aquellas que permiten especificar explícitamente un nombre de pantalla. Mediante `bitepilogue` se cierra una pantalla. Con `bitmap-save` se puede guardar para ser restaurada posteriormente con `bitmap-restore`. La función `bitmap-flush` se utiliza en sistemas con buffer, como X Window System, para indicar explícitamente que se realicen las operaciones almacenadas en el buffer. Mediante funciones VBD se puede obtener información concerniente a la resolución, los colores disponibles, las texturas (*patterns*), las fuentes, el cursor, etc.

Ventanas

Las ventanas son estructuras que tienen asociadas representaciones gráficas en la pantalla. Cada ventana tiene asociada un entorno gráfico que almacena información concerniente al cursor, los colores, la fuente, la textura, el estilo de línea, el modo de dibujo y la zona de *clipping* actualmente utilizados en esa ventana. Los eventos asociados a las ventanas pueden estar causados por movimientos o *clicks* del ratón o por operaciones realizadas por el *gestor de ventanas*. Una ventana puede contener subventanas, las cuales a su vez pueden contener más subventanas, lo cual permite establecer una jerarquía. Todas las operaciones que se pueden realizar sobre ventanas son aplicables también a las subventanas.

Fuentes

Indican cómo se van a representar los caracteres en pantalla. EN VBD, una fuente se referencia mediante un entero pequeño. Mediante llamadas a `load-font` se pueden cargar fuentes disponibles en el sitema, teniendo en cuenta que el string que se le pasa como argumento varía según el sistema usado y por tanto no es portable. En X11 tendría un aspecto así:

```
-adobe-helvetica-bold-r-normal--12-120-75-75-p-82-iso8859-1
```

Mediante `font-max` se puede conocer el número máximo de fuentes disponibles y mediante `current-font` se puede establecer la fuente actual.

Colores

El color de primer plano o *foreground* se establece mediante `current-foreground` y se utiliza para dibujar las líneas y los caracteres y para el relleno de zonas cuando la textura de relleno es `standard-foreground-pattern`. El color de fondo o *background* se establece mediante `current-background` y se utiliza como color de fondo en los caracteres y para el relleno de zonas cuando la textura actual es `standard-background-pattern`.

Mediante `make-color` se puede crear un nuevo color indicando los componentes RGB. La función `make-named-color` devuelve un color cuyos componentes RGB están especificados en la base de datos de color del sistema. Mediante `make-mutable-color` podemos crear colores cuyos componentes RGB se pueden cambiar dinámicamente, lo cual es útil para conseguir efectos de parpadeo y animación. Cuando un color no se va a utilizar más, es conveniente llamar a `kill-color`.

Cursores

El cursor es la imagen del ratón sobre la pantalla. En LE-LISP se referencian mediante un entero pequeño. Mediante `current-cursor` se cambia el cursor actual. Utilizando `make-cursor` es posible crear un cursor a partir de un bitmap. Existe un conjunto de cursores definidos en el sistema a los que se puede acceder llamando a `make-named-cursor`.

Texturas

Para rellenar áreas se usan texturas (*patterns*) de uno o de dos colores. Se pueden crear texturas a partir un icono. El relleno se efectuará repitiendo regularmente la imagen del icono. Funciones que tratan con texturas son `current-pattern`, `pattern-max` y `make-pattern`.

Modos de dibujo

El modo en que se combinan los puntos en la pantalla cuando se realizan dibujos se puede establecer mediante la función `current-mode`. Hay 16 modos disponibles para combinar un pixel de la imagen a dibujar con los pixels ya presentes en la pantalla.

Primitivas gráficas

Las funciones `draw-polyline`, `draw-polymarker`, `fill-area` y `draw-substring` implementan las cuatro primitivas de menor nivel definidas en el estándar GKS [Hopgood et al. 83]. Las primitivas para dibujar objetos elipsoidales se implementan mediante `draw-ellipse` y `full-ellipse`. Por razones de eficiencia, LE-LISP proporciona funciones adicionales de más alto nivel[13] que incluyen el dibujo de puntos, línea, rectángulos, círculos y arcos circulares, así como el relleno de zonas rectangulares y circulares,incluyendo sectores circulares. También se proporcionan funciones adicionales para el manejo de texto.

Bitmaps

Un mapa de bits es un array bidimensional de bits que se implementa mediante la estructura `bitmap`. Actualmente el VBD tan sólo es capaz de manipular bitmaps de dos colores. La modificación y consulta de las características de un bitmap se realiza por medio de métodos asociados a la estructura. Una característica interesante es la posibilidad de comprimir un bitmap hasta en un factor de 5 a 1. La compresión se realiza siempre que la variable `#:system:compressed-icon` es distinta de `()`, con la facilidad de que la lectura de los bitmaps siempre se realiza correctamente ya que se determina dinámicamente si la fuente está comprimida o no.

Los bitmap se crean mediante `create-bitmap` y se puede liberar la memoria que ocupan cuando no se utilizan llamando a `kill-bitmap`. Se pueden examinar y modificar bits individuales mediante `bmref` y `bmset`, respectivamente. Mediante `bitblit` se pueden copiar zonas rectangulares de un bitmap fuente a otro destino[14].

Bytemaps

Un mapa de bytes es un subtipo de bitmap sin campos adicionales que permite manipular hasta 256 colores. Se crean mediante `create-bytemap` y se manipulan mediante funciones similares a las utilizadas con los bitmaps así como otras que manipulan los colores.

B.1.7 Virtual mouse

El *ratón virtual* es el método mediante el cual se gestiona un dispositivo que selecciona puntos discretos sobre una pantalla gráfica. Este dispositivo virtual puede corresponderse en el plano físico con un ratón, un lápiz óptico, una tableta gráfica, una pantalla táctil o cualquier otro dispositivo apuntador.

Eventos

Un evento es la unidad básica de información generada por el VBD y puede estar causado por una acción *física* del dispositivo apuntador sobre una ventana, por la pulsación de una tecla, por una modificación del estado de una ventana causada por el sistema gestor de ventanas o por una petición de refresco de una ventana.

[13] Que pueden ser descritas a partir de las primitivas GKS.
[14] Que no tienen por qué ser distintos.

Cada sistema genera un conjunto de eventos diferente, sin embargo, LE-LISP proporciona una lista de tipos de evento lo suficientemente amplia para trabajar con los sistemas gráficos más conocidos.

Los eventos son instacias de la estructura `event` y van siendo almacenados en la *cola de eventos*. La estructura `event` contiene información sobre el tipo de evento, la ventana afectada, las coordenadas globales y locales donde se produjo y un campo `detail` que contiene información adicional y depende del tipo de evento del que se trate.

Los tipos de evento con los que nos enfrentaremos más a menudo son:

- **down event**, que indica la pulsación de un botón del ratón. El código del botón concreto pulsado así como la indicación de si se realizó un doble click o una pulsación sencilla se almacenan en el campo `detail`.
- **move-event**, que indica un movimiento del dispositivo apuntador.
- **ascii-event**, producido al pulsar una tecla. El código ASCII correspondiente a dicha tecla se almacena en `detail`.
- **modify-window-event**, que indica que un agente externo a LE-LISP, generalmente el gestor de ventanas, ha modificado la posición o el tamaño de una ventana.
- **kill-window**, que indica una petición de destrucción de una ventana por parte de un agente externo.
- **enterwindow-event**, que junto con **leavewindow-event** permite controlar cuando el ratón entra y sale de una ventana.
- **repaint-window-event** significa que el sistema necesita refrescar una ventana.
- **keyboard-focus-event** indica un cambio en la propiedad del teclado por parte del sistema.
- **visibility-change** indica que han sido modificadas las condiciones de visivilidad de una ventana[15]. El evento **map-window** indica que una ventana a pasado a primer plano y **unmap-window** que ha desaparecido de la pantalla o que ha sido iconificada.

La manera en que estos eventos se añaden a la cola depende del *modo* en que se encuentre el ratón, que viene indicado por el estado de ciertos *flags*. Con el *abbreviated mouse mode flag* activo, cuando ocurre un move-event, éste se añade a la cola a menos que el último evento en ella sea un move-event o un drag-event, en cuyo caso caso lo reemplazaría. Con el *mouse interrupt mode flag* activo, antes y después de que el evento sea añdido a la cola, se produce una interrupción programable llamada `event`. La función `event` permite establecer la función encargada de manejar los eventos cuando se produce esta interrupción. El modo activo en cada momento se puede establecer mediante la función `event-mode`. Inicialmente ninguno de los dos flags está en *on*.

La cola de eventos se puede leer mediante `read-event`, que implica la eliminación del evento leído, y `peek-event`, que examina el evento pero no lo elimina. Mediante `flush-event` se puede vaciar toda la cola de eventos, mientras que `add-event` permite añadir un evento.

[15] Por ejemplo, que ha sido parcialmente cubierta por otra.

Llamando a `grab-event` se puede convertir a una ventana en propietaria de todos los eventos de ratón y teclado. En los sistemas donde el ratón es compartido por varios programas, esta función también convierte al LE-LISP en propietario del ratón. Se suele utilizar preferentemente cuando se utilizan pantallas de aparición súbita para realizar la cofirmación de alguna acción.

Una función muy utilizada es `read-mouse`, que devuelve una instancia de la estructura event que describe la posición y el estado actal del ratón.

Mediante `create-menu`, `activate-menu`, `kill-menu` y un conjunto de funciones destinadas a establecer los items de las listas de selección, se pueden construir menús con opciones seleccionables por el usuario.

Cortar y pegar

La función `display-store-selection` toma una cadena de caracteres y la copia en un buffer asociado con la pantalla. Si el dispositivo lo permite, se puede compartir dicha cadena con todas las aplicaciones que usan esa pantalla.

La función `display-get-selection` recupera la cadena de caracteres que constituye la selección actual en una pantalla. En sistemas como X Window System esta función permite recuperar una cadena proporcionada por cualquier otra aplicación que usa el mismo X-server.

B.2 AÏDA

Sobre LE-LISP se han construído un variado conjunto de herramientas, una de las cuales es AÏDA, creada por la empresa francesa ILOG[16] para facilitar la construcción de interfaces gráficos en entornos de ventanas. Los programas en AÏDA son programas LE-LISP, con la salvedad de que se puede hacer uso de la extensa biblioteca de objetos (estructuras) incorporada. Es precisamente la disponibilidad de este conjunto de objetos de alto nivel lo que facilita la tarea, ya que permite diseñar construcciones gráficas avanzadas sin tener que recurrir continuamente a las funciones de bajo nivel implementadas en el VBD[17]. Con esto se consigue, por ejemplo, abrir y controlar un elevado número de ventanas, cada una conteniendo elementos gráficos diferentes y con comportamientos distintos ante las acciones del ratón, del teclado y del gestor de ventanas, sin tener que realizar explícitamente las tediosas y rutinarias tareas involucradas en la creación y control de eventos de cada una de ellas, ya que dispondremos de objetos y métodos de mayor nivel que nos permitirán *filtrar* aquellos comportamientos que deseemos manejar explícitamente, permitiendo que el resto actúen adecuadamente mediante los mecanismos estándar definidos para cada objeto. Podemos considerar por tanto, que así como LE-LISP proporciona un medio de abstracción con respecto al hardware subyacente, AÏDA supone una abstracción con respecto al LE-LISP mismo, permitiendo que el programador se centre en los aspectos de diseño y comportamiento de la interfaz, contando con la seguridad de un correcto funcionamiento en las tareas de inferior orden, en cuanto al nivel de abstracción. Sin embargo, siempre que sea preciso, se tiene un acceso total a todas

[16] **I**nteligence **L**ogicielle.
[17] **V**irtual **B**itmap **D**isplay

las funciones disponibles en LE-LISP. Además, AÏDA dispone de un conjunto bastante completo de herramientas de ayuda al desarrollo que abarcan desde un sistema de ayuda con ejemplos sobre los que experimentar hasta un completo sistema de inspección de objetos.

A continuación se va a realizar una breve introducción a las características y herramientas de AÏDA. No se trata de una descripción completa, sino de un breve sumario que permita al lector comprender mejor los aspectos de la implementación presentada en esta tesina. Para aquellas personas que estén interesadas en un estudio más en profundidad, lo mejor es comenzar por [ILOG 92e] y proseguir por [ILOG 92b]. Como es natural, el punto de referencia ante cualquier duda o para profundizar en algún aspecto concreto es [ILOG 92c].

B.2.1 Imágenes

Desde el punto de vista de AÏDA, una imagen es un objeto LE-LISP que puede ser dibujado en una pantalla gráfica. AÏDA proporciona un conjunto de objetos que implementan imágenes atómicas[18] y compuestas[19], así como un *lenguaje de descripción de imágenes* que permite construir nuevas imágenes mediante la aplicación de *constructores de imágenes* a las ya definidas.

Las imágenes se representan en LE-LISP mediante estructuras capaces de responder a mensajes como:

- `display`, que indica una petición para dibujar la imagen en pantalla. La petición puede ser para la imagen completa o para una parte o *región* de la misma.
- `invert-display`, que muestra una imagen en video inverso.
- `bounding-box`, que se utiliza para obtener el menor rectángulo capaz de contener a la imagen.
- `width`, que devuelve la anchura de la bounding-box.
- `height`, que devuelve la altura de la bounding-box.
- `x` e `y` devuelven, respectivamente, las coordenadas horizontal y vertical del *display point*, que se corresponde con la esquina de la bounding-box más cercana al eje de coordenadas. Siempre que se dibuja una imagen en pantalla, se utiliza el *display point* como punto de referencia.

También es posible modificar físicamente las imgenes. Para ello se hace uso de unos métodos llamados *transformaciones*. Con ello se evita la creación o asignación de imágenes completamente nuevas. Sin embargo, las transformaciones no suelen tener un efecto inmediato en pantalla, sino que debe ser el propio programador el que indique explícitamente que se redibujen ciertas imágenes. Como precaución hay que tener en cuenta que hay imágenes que no son modificables físicamente; en tales casos el resultado correcto está en la imagen devuelta, ya que la imagen argumento no habrá sufrido cambios. Transformaciones muy utilizadas son:

[18] Aquellas que no se pueden descomponer a su vez en imágenes más simples.
[19] Formadas por la combinación de imágenes atómicas.

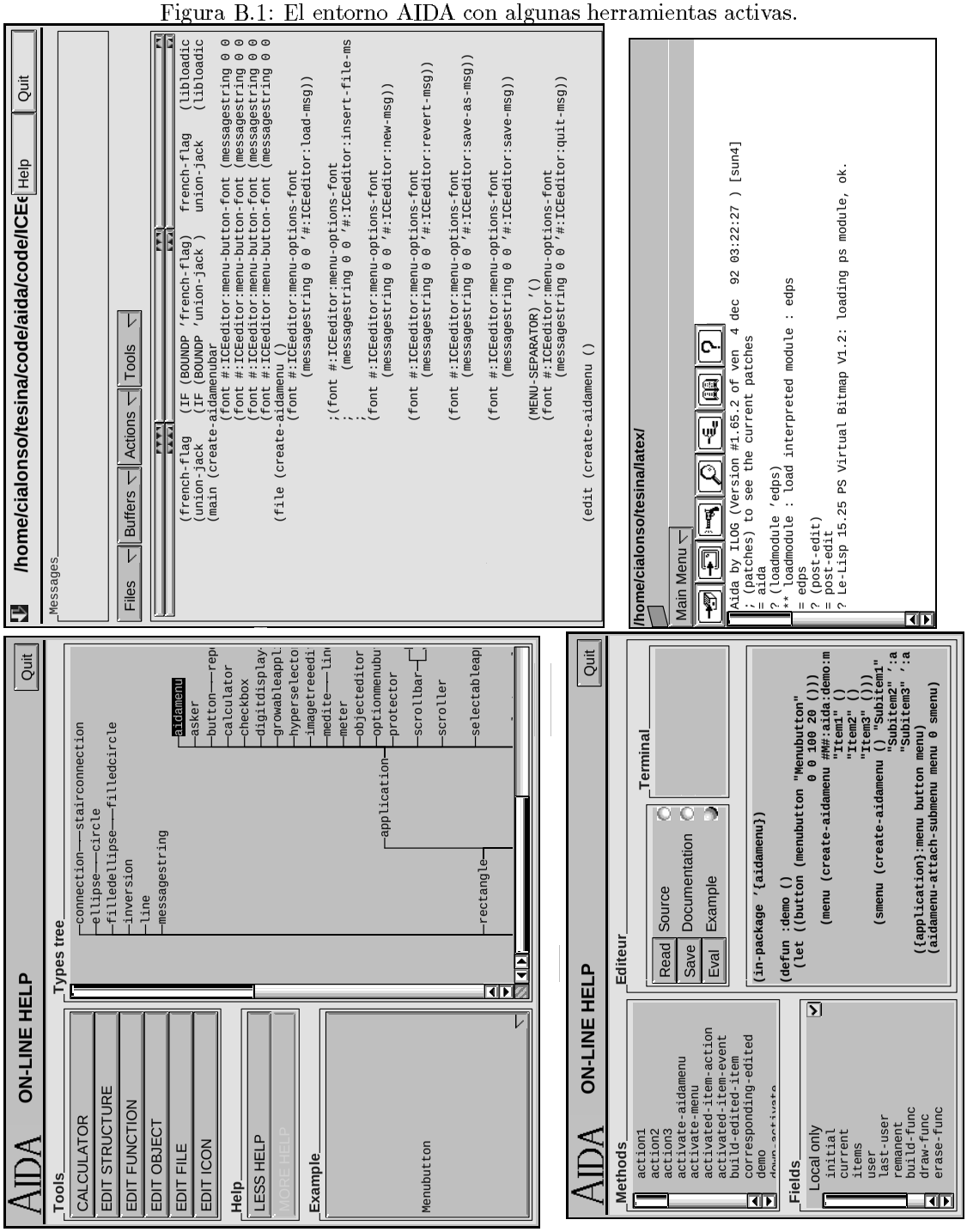


Figura B.1: El entorno AÏDA con algunas herramientas activas.

- `translate`, utilizada para desplazar la posición de la imagen en la pantalla. No es aplicable sobre cadenas de caracteres.
- `grow`, que se utiliza para modificar las dimensiones de la imagen. Los iconos y las cadenas de caracteres no pueden modificar su tamaño.
- `resize`, prácticamente igual que `grow` excepto que se puede indicar que la imagen sólo crezca (o mengüe) en una dimensión.

Otros métodos interesantes son `intersecp`, un predicado que indica si un punto en una imagen está localizado dentro de una determinada sección rectangular de la pantalla, y `map-constituents`, que permite aplicar una función a todas las imágenes que a su vez componen la imagen, aunque tan solo en un primer nivel de descomposición.

AÏDA también proporciona un conjunto de macros que facilitan la definición del tamaño de las imágenes: Estas macros son:

- `#<tanto-por-ciento>wdisplay` o `#<tanto-por-ciento>wd`, que devuelven un entero que representa la porción del ancho de la pantalla indicada por el entero `<tanto-por-ciento>`.
- `#<tanto-por-ciento>hdisplay` o `#<tanto-por-ciento>hd`, como la anterior, pero para la altura.
- `#<número>wchar` o `#<número>wc` devuelve el ancho en pixels del espacio en blanco en la fuente actual multiplicado por `<número>`.
- `#<número>hchar` o `#<número>hc`, como la anterior pero para la altura.

B.2.2 Los constructores de imágenes

Se utilizan para crear imágenes y para combinarlas entre sí. Devuelven instancias de estructuras LE-LISP que representan las imágenes construidas. A todas ellas se les pueden aplicar los mensajes `display`, `x`, `y`, `width`, `height`, `bounding-box`, `grow` y `translate`. Existen métodos adicionales específicos para cada tipo de imagen, que están asociados a las estructuras que las representan.

Rectángulos

Son instancias de `#:image:rectangle`, abreviadamente {`rectangle`}, una estructura con 4 campos llamados `x`, `y`, `w` y `h` que representan respectivamente las coordenadas horizontal y vertical, el ancho y alto del rectángulo. Se utilizan mucho en AÏDA ya que entre otras cosas sirven para almacenar la bounding-box de cualquier imagen. Cabe señalar que los rectángulos son imágenes transparentes en el sentido de que no producen ningún efecto visible en pantalla. Si lo que se quiere es dibujar las líneas del borde de un rectángulo, deberán utilizarse imágenes del tipo **cajas rectángulares**.

Imágenes geométricas

- **Cajas rectángulares**. Muestran las líneas de los bordes de un rectángulo.

- **Cajas coloreadas**[20]. Una zona rectangular dibujada en un color determinado.
- **Cajas rectángulares rellenas**[21]. Como la anterior sólo que se puede utilizar una textura definida en LE-LISP.
- **Elipses**.
- **Elipses rellenas**.
- **Círculos**.
- **Círculos rellenos**.
- **Triángulos**.

Cadenas de caracteres

- **Cadenas de caracteres**. No se necesita una estructura ni una función de creación para ellos ya que son cadenas de caracteres normales de LE-LISP. Se dibujan usando la fuente actual y no pueden variar de tamaño ni ser trasladados.
- **Cadenas de caracteres multi-idioma**. Es la imagen AÏDA que representa el valor de los mensajes multi-idioma de LE-LISP. Su imagen cambia dinámicamente según sea el idioma utilizado. Se utiliza la estructura `#:image:messagestring` para representarlos, lo que permite que a diferencia de las cadenas de caracteres convencionales, las multi-idioma sí puedan ser trasladadas.
- **Fuentes**. Permite que las cadenas de caracteres dentro de una imagen sean dibujadas en una fuente determinada.

Iconos

- **Iconos**. Son imágenes rectángulares que contiene un bitmap. Las dimensiones del mapa de bits determinan el tamaño del icono. Si se desea que tenga más de un color se debe usar un bytemap en lugar de un bitmap en el momento de crearlo.
- **Iconos elásticos**. A diferencia de los iconos normales, éstos pueden modificar su tamaño.
- **Iconos transparentes**. En ellos los pixels con color correspondiente al *standard background* se comportan como si fuesen transparentes.

Superposición de imágenes

- **Superposiciones**[22]. Permiten dibujar un conjunto de imágenes superpuestas. Cuando se traslada una superposición, se aplica el cambio de posición a cada imagen componente. Sin embargo, una superposición no puede cambiar de tamaño.

[20] *Colored boxes.*
[21] *Filled rectangular boxes.*
[22] *Image overlays.*

- **Superposiciones centradas**. Las imágenes se sitúan una encima de otras de tal modo que los centros de sus bounding-box respectivas coinciden en el mismo punto.
- **Superposiciones elásticas**. Como una superposición normal salvo que las imágenes crecen y se mueven proporcionalmente cuando se aplica un método `grow` o `translate`.
- **Superposiones centradas elásticas**. Una combinación de los tipos precedentes.
- **Columnas**. Permiten colocar un conjunto de imágenes encolumnadas. Cuando una columna recibe un mensaje `grow` con argumentos `w` y `h`, la anchura de cada componente se modifica a `w`. Si la nueva altura es mayor que la anterior, el último componente crece hasta llenar el nuevo espacio. Si por el contrario la nueva altura no es suficiente para acomodar todos los componentes con su altura original, la altura final es la suma de las alturas de todos los componente excepto el último.
- **Columnas elásticas**. Como las anteriores, salvo que las imágenes crecen proporcionalmente cuando se modifica el tamaño de la columna.
- **Filas**. Un conjunto de imágenes colocadas en fila. Recíprocamente a lo que sucede con las columnas, cuando a una fila se le envía el mensaje `grow` con argumentos `w` y `h`, la altura de cada componente se establece en `h`. Si la nueva anchura es mayor que la existente, el último componente se alarga hasta ocupar todo el nuevo espacio. Si es menor, la anchura final será la suma de las anchuras de todos los componentes menos el último.
- **Filas elásticas**. Como las anteriores salvo que el tamaño de las imágenes varía proporcionalmente cuando se modifica el tamaño de la fila.
- **Vistas restringidas**[23]. Son más flexibles que las anteriores. Permite agrupar imágenes en la pantalla definiendo individualmente cómo se comportará cada una de ellas cuando la imagen sea movida o varíe de tamaño. Para ello se define una *restricción* para cada imagen que establece sus *puntos de anclaje* respecto a los bordes de la ventana.
- **Imágenes enmarcadas**[24]. Es el resultado de superponer una caja rectángular sobre una imagen centrada.
- **Imágenes enmarcadas elásticas**. Como la anterior salvo que la imagen crece proporcionalmente cuando varía el tamaño.

Adornos

- **Cruces**[25].
- **Clusters**. Un rectángulo con un título. Se utilizan para organizar las ventanas en zonas con información relacionada.
- **Encabezamientos** y **clusters**. Son vistas restringidas especialmente diseñadas para adornar una ventana.

[23] *Constrained views.*
[24] *Boxed images.*
[25] *Crosses.*

Imágenes unarias

Mediante ellas se pueden definir nuevos tipos de imágenes en AÏDA basándose en la idea de una imagen que contiene a otra imagen. Este tipo de imagen permite compartir los métodos de todas las imágenes que contienen otra imagen. Un objeto de tipo unario reacciona a todos los mensajes transmitiendo el mismo mensaje a la imagen que contiene. Se usa, por ejemplo, en los tipos **modo**, **textura** y **estilo de línea**.

Otras imágenes

- **Conexiones binarias**. Permiten dibujar enlaces entre imágenes. Son muy útiles cuando se trata de construir una herramienta para dibujar diagramas.
- **Conexiones unidas**[26]. Se diferencian de las anteriores solamente en la forma en que son dibujadas.
- **Imágenes trasladadas**. Es el resultado de dibujar una imagen en un sistema de coordenadas que ha sufrido una traslación del origen. La imagen conserva su forma y orientación.
- **Imágenes centradas**. Permiten centrar una imagen, tanto horizontal como verticalmente, dentro de una zona rectangular.
- **Imágenes invertidas**. Es el resultado de aplicar la operación de vídeo inverso a cada pixel de la imagen.
- **Modos**. Muestra una imagen en un modo del VBD específico.
- **Texturas**[27]. Construye una imagen utilizando una textura determinada.
- **Estilos de líneas**. Permite construir imágenes con un determinado estilo de línea.
- **Color de dibujo**[28]. Permite construir una imagen con un determinado color de dibujo.
- **Color de fondo**[29]. Permite construir una imagen con un determinado color de fondo.
- **Plano de fondo coloreado**[30]. Construye una imagen que es dibujada sobre un plano de fondo coloreado. Es muy útil en ciertos casos, cuando el constructor precedente no ofrece los resultado deseados.

B.2.3 Aplicaciones

Las aplicaciones son instancias de `#:image:rectangle:application`, abreviadamente {`application`}. Esto quiere decir que una aplicación es también una imagen de tipo rectángulo, lo cual implica que por sí misma no es visible en pantalla, tan sólo delimita

[26] *Jointed connections.*
[27] *Patterns.*
[28] *Foreground.*
[29] *Background.*
[30] *Colored background plane.*

una región de ésta. Sin embargo, la estructura {`application`} contiene un campo llamado `image` que almacena una imagen. Es el contenido de este campo lo que se muestra en pantalla cuando se *visualiza* una aplicación.

Como consecuencia de lo anterior, ya que una aplicación es una imagen, el campo `image` puede a su vez contener una aplicación, la cual a su vez puede contener aplicaciones hasta cualquier nivel de profundidad.

Desde un punto de vista práctico, podemos ver una aplicación como una imagen a la cual se le asocia un *comportamiento*, esto es, un conjunto de acciones que tiene lugar como *reacción* de la aplicación ante operaciones llevadas a cabo con el ratón o el teclado. Los diferentes subtipos de aplicaciones presentan diferentes comportamientos ante esas operaciones. Es incluso posible especificar el comportamiento concreto de cada instancia de un determinado tipo de aplicación.

Como cualquier otra imagen, las aplicaciones se crean utilizando un constructor de imágenes, en este caso llamado `application`, que devuelve una instancia de la estructura {`application`}, la cual además del campo `image` también posee un campo `window` que almacena la ventana asociada a la aplicación[31] y un campo {`properties`} que contiene una lista de asociación de propiedades.

Mensajes

Las aplicaciones reaccionan a los siguientes mensajes que modifican su imagen:

- El mensaje `grow` produce una modificación del tamaño de la ventana asociada a la aplicación. Para ello se envía resursivamente un método `grow` a las imágenes contenidas en la aplicación.

- El mensaje `translate` provoca un traslado de la ventana de la aplicación que no afecta a la imagen. Este método, como el anterior, puede ser llamado explícitamente por el programador o puede ser activado como consecuencia de una modificación de las características de la ventana que contiene la aplicación provocadas por acciones del usuario valiéndose del gestor de ventanas.

- El mensaje `fit-to-contents` indica a la aplicación que ajuste su tamaño para que sea consistente con el de la imagen que contiene. Este método debe ser enviado antes de que la aplicación esté activa en la pantalla, ya que de lo contrario no surtirá efecto.

- El mensaje `fit-to-window` hace que la imagen de una aplicación adapte su tamaño al de la ventana.

- El mensaje `redisplay` indica a la aplicación que debe redibujar una región de la imagen. Puede ser activado por el programador o por el gestor de ventanas.

- El mensaje `full-redisplay` borra la ventana asociada con una aplicación e inmediatamente la redibuja enviando un mensaje `redisplay`.

- El mensaje `new-image` cambia la imagen de una aplicación, modificando el campo `image`, pero sin redibujarla en pantalla.

[31] Cuyo tipo es {`aidawindow`}, definido en el VBD.

- El mensaje `set-image` cambia la imagen de una aplicación y la redibuja en pantalla.
- El mensaje `add-image` añade un componente a la imagen de la aplicación, que debe ser una fila, columna o superposición.
- El mensaje `remove-image` elimina un componente de la imagen.
- El mensaje `insert-image` inserta un componente en la imagen de la aplicación asignándole una posición en el rango de imágenes que la componen.
- Los mensajes `add-to-left` y `add-to-right` añaden componentes a la izquierda y derecha de la imagen, respectivamente.
- Los mensajes `add-to-top` y `add-to-botton` añaden componentes sobre la imagen o al fondo de la misma.

Los métodos `display`, `redisplay` y `grow` no se deben redefinidir para subtipos de {`application`}, ya que su comportamiento no sería correcto. Si se necesita redefinirlos, la solución consiste en crear un nuevo subtipo de {`image`} que represente la imagen de la aplicación y redefinir los métodos mencionados para la nueva imagen.

Funciones que tratan con aplicaciones

- `activatedp` es un predicado que indica si una aplicación está activa en la pantalla.
- {`application`}:`father` devuelve la aplicación padre de una dada, es decir, aquella en la que está contenida.
- `set-action` es probabalemente la función más usada con aplicaciones ya que establece la función que se realizará cuando se *dispare* la aplicación. Cada tipo de aplicación se dispara ante determinados eventos; por ejemplo, un botón se dispara cuando es pulsado con el ratón.
- `set-shortcut-action` permite asociar una segunda acción con una aplicación. Algunas aplicaciones realizan una segunda acción en respuesta a ciertos actos del usuario. Por ejemplo, en una lista de selección la shortcut-action tiene lugar cuando el usuario hace click con el ratón sobre una de las imágenes seleccionables.
- `add-application` activa una aplicación y sus subaplicaciones. El efecto que se ve en la pantalla es la aparición de una nueva ventana con la imagen asociada. Si el argumento que se le pasa a esta función es una imagen en lugar de una aplicación, entonces crea una instancia de {`application`} con dicha imagen como valor del campo `image` y la activa en pantalla.
- `remove-application` termina una aplicación y todas sus subaplicaciones, eliminándolas de la pantalla.
- `prompt-application` actúa como `add-application` excepto que no devuelve el control hasta que no se ha terminado la aplicación. Se usa principlamente para construir ventanas de confirmación.

- `grab-application` actúa como `prompt-application` excepto que impide al usuario interactuar con otras aplicaciones hasta que termine la aplicación lanzada. Se usa para construir ventanas de confirmación de acciones muy relevantes.

- `inhibit-application` impide que una aplicación reaccione a los eventos del ratón o del teclado. Mediante `authorize-application` se puede hacer que dicha aplicación retorne a un comportamiento normal.

- {`application`}:`set-title` establece el título de la ventana que contiene la aplicación, si el gestor de ventanas lo permite.

Subaplicaciones

Como ya se mencionó anteriormente, la imagen de una aplicación puede contener a su vez otras aplicaciones, las cuales se consideran subaplicaciones de la primera. De este modo es posible establecer una jerarquía de aplicaciones de múltiples niveles. Para permitir una adecuada comunicación entre la aplicación *principal* y sus subaplicaciones y de éstas entre sí, AÏDA dispone de un mecanismo basado en la idea de dar nombre a los componentes.

Para ello basta con definir una subaplicación presente en la imagen de otra como un componente de esta última, asignándole un nombre. Esto se hace mediante la función `add-component`. Una vez establecido, un componente puede cambiarse mediante la utilización de `set-component`, o eliminarse mediante una llamada a `remove-component`. Para acceder a una subaplicación componente de otra desde cualquier componente de esta última se usa la función `component`. Mediante una llamada a `all-componentes` se obtiene una lista de los componentes definidos al mismo nivel, o a uno superior, de la aplicación pasada como argumento.

B.2.4 Aplicaciones de AÏDA

En la jerarquía de objetos incorporada en AÏDA se encuentran definidas un elevado conjunto de aplicaciones que se pueden utilizar como componentes de la aplicación que se desea construir. A pesar de que muchas de esas aplicaciones permiten un cierto nivel de adaptación, algunas veces puede ser necesario definir nuevos subtipos a partir de los existentes o cambiar el comportamiento de ciertas instancias ante distintos eventos.

A continuación se muestra una lista de la mayor parte de las aplicaciones predefinidas.

Botones

- **Botones**. Implementan los *push-buttons*. Hay dos tipos: los botones normales creados mediante la función `button` y otros similares pero rodeados de una *buton-box* y que se crean usando `standardbutton`

- **Check boxes**. Registran el estado de una variable de dos estados. Su imagen generalmente consta de una cadena de caracteres seguida de un icono en forma de caja, que contendrá una marca si ha sido seleccionada. Se crean mediante la función `checkbox`.

- **Botones de radio**. Se utilizan para registrar el estado de un parámetro de dos estados, indicando si el valor es válido o no. Se dibuja generalmente como una cadena de caracteres, seguida de un icono que generalmente representa un botón circular. Los botones de radio se crean mediante `radiobutton` y son imágenes seleccionables, por lo que que pueden ser utilizados directamente en un selector.
- **Botones con menú**. Son botones que muestran un menú cuando el usuario pulsa con el ratón sobre él. La opción del menú que sea seleccionada se convertirá en la etiqueta del botón. El menú no puede contener aplicaciones AÏDA[32]. Se crean mediante la función `optionmenubutton`.
- **Botones con menú y valores**. Son similares a los anteriores salvo que también asocian una lista de valores con la lista de imágenes. Se crean mediante `valuemenubutton`.

Menús

En entornos como X11 hay que tener en cuenta que el padre de las ventanas *pull down* se asigna a la ventana raiz[33] para permitir que un menú se pueda seguir utilizando aun cuando las dimensiones de la ventana de la aplicación a la que pertenece no sean suficientes para contener el menú desplegado.

Los menús creados con `create-aidamenu` pueden contener en sus opciones cualquier imagen, mientras que aquellos cuya creación se ha realizado mediante `stringmenu` sólo muestran strings. Estos últimos tienen un comportamiento más próximo al de los selectores que al de los menús propiamente dichos.

Desplazadores

Mediante las funciones `verticalscroller` y `horizontalscroller` se puede encerrar una imagen en una caja con una barra de desplazamiento horizontal o vertical. Si se desean ambas se deberá utilizar `scroller`. Se dispone de un conjunto de métodos para fijar las características de las barras de desplazamiento y para provocar un desplazamiento de la imagen.

Si se desea disponer de barras de desplazamiento separadas de la imagen, se pueden definir mediante `verticalscrollbar` y `horizontalscrollbar`. El aspecto que presentan en pantalla no se corresponde con el *look* de ningún gestor de ventanas específico como pueda ser Motif o OpenLook, sino que utiliza widgets[34] propios de AÏDA.

Selectores

Permiten seleccionar con el ratón una o varias imágenes de entre un conjunto de ellas. Cuando el conjunto de imágenes seleccionables es muy elevado se suele recurrir a un scroller para hacer más manejable el selector. Las imágenes suelen estar organizadas en forma de fila o columna, aunque la única restricción impuesta es que deben ser imágenes seleccionables. Cualquier imagen puede convertirse en selecccionable si se pasa como

[32] Pueden contener cualquier imagen que no sea una aplicación.
[33] La correspondiente al fondo de la pantalla.
[34] Bloques de pantalla estándar que permiten unificar el aspecto de un conjunto de aplicaciones.

argumento a la función `selecteableapplication`. Un subtipo especial lo componen las aplicaciones creadas mediante `greyselectable`, que se diferencian de las anteriores en que cuando no están seleccionadas su imagen se muestra en gris. Los selectores se crean pasando el conjunto de imágenes seleccionables a `selectorapplication`.

Un caso especial de selectores lo constituyen los hiperselectores, que se crean mediante `hiperselector` y cuya característica distintiva es que permite filtrar el conjunto de strings[35]. El filtrado se consigue mediante los caracteres que el usuario teclea en el editor de líneas que los hiperselectores llevan acoplado en la parte inferior.

Editores

AÏDA incorpora una serie de editores básicos que pueden servir de base para la elaboración de otros más sofisticados. El más completo es el correspondiente a la aplicación `medite`, ya que se trata de un editor multilínea que admite la edición del texto mediante combinaciones de teclas similares a las utilizadas por el popular Emacs. Dispone de una buena interfaz para el programador ya que proporciona un conjunto bastante amplio de métodos que permiten un control eficaz del texto. Sin embargo presenta una importante limitación: sólo permite utilizar una única fuente y un color.

Si se precisa utilizar un editor que permita cambiar la fuente en porciones del texto, admitiendo una mezcla libre de fuentes de anchura fija y proporcional y en el que se pueda colorear porciones de texto, deberemos recurrir a `textedit`, un editor que se proporciona como extensión estándar de AÏDA. Otra ventaja que presenta es la de poder insertar imágenes, incluso aplicaciones, entre el texto. Sin embargo, tiene como inconveniente un manejo más complicado y ciertas carencias que obligan, por ejemplo, a definir manualmente el movimiento de scroll horizontal.

Para la edición de una sóla línea se dispone de un subtipo de `medite` denominado `lineedit`. Presenta problemas cuando se intenta utilizar con una fuente proporcional, ya que no es capaz de manejar adecuadamente el cursor.

Este problema se soluciona utilizando `stringedit`, que no es un subtipo de los anteriores. Se caracteriza porque sus instancias ocupan muy poco espacio en memoria y porque permite un filtrado de los caracteres tecleados y maneja perfectamente las fuentes proporcionales. Se utiliza asiduamente cuando es necesario presentar al usuario una línea para introducir algún tipo de dato, como por ejemplo el nombre de un fichero.

Como un subtipo del anterior tenemos `formatedit` que pemite definir el formato de la entrada, por lo cual es muy útil para el relleno de formularios.

Como un tipo especial de editor podemos considerar a los *terminales virtuales*, creados mediante llamadas a `ttyapplication`. Son áreas rectángulares de la pantalla en la que el usuario puede dialogar con AÏDA escribiendo expresiones en LE-LISP.

Los editores definidos en AÏDA son capaces de trabajar con caracteres acentuados. Para ello se debe utilizar una fuente que los incluya en su juego de caracteres y activar el modo acentuado evaluando `(aida-manage-accent t)`. El mecanismo para acentuar interactivamente los caracteres es peculiar, puesto que es preciso teclear el apóstrofe ' antes de la letra que se desea acentuar, al contrario de lo que secede normalmente. Otro mecanismo consiste en redefinir las teclas de función de modo que su pulsación represente un caracter acentuado.

[35] Sólo admiten cadenas de caracteres como imágenes seleccionables.

Árboles

Para conseguir objetos móviles se utilizan las *imágenes móviles*, que se crean mediante `moveableapplication` y son básicamente imágenes seleccionables que se pueden mover mediante el ratón. Cuando una de estas imágenes es movida por el usuario, una *imagen fantasma* sigue el movimiento del cursor del ratón hasta que se libera el botón correspondiente.

Un tipo especial de imágenes móviles lo constituyen las *imágenes nodo*, utilizadas para implementar los nodos en los editores de árboles. Estos últimos se crean mediante `treeeditor` y permiten editar tanto árboles propiamente dichos como grafos acíclicos.

Si se desea un editor de grafos más general y sofisticado se debe recurrir al `grapher`, que constituye una de las extensiones estándar de AÏDA.

Cajas de diálogo

Permiten crear ventanas con un cierto número de botones e imágenes. Mediante la simple pulsación de un botón, el usuario debe poder responder a la cuestión que planteó la necesidad de mostrar la caja de diálogo. AÏDA proporciona un conjunto de cajas de diálogo que cubre una amplia variedad de posibilades:

- **confirmadores**, que incorporan de serie un botón de OK y otro de Cancel.
- **indicadores de advertencia**. Tan sólo tiene el botón de OK y se utilizan para advertir al usuario de alguna situación especial.
- **peticiones de string**, como la anterior, pero con un editor de strings que permite que el usuario teclee, por ejemplo, un nombre de fichero.
- **peticiones de objetos**, permiten al usuario elegir uno o varios objetos de entre una lista mostrada en un selector.

Adicionalmente se dispone de un conjunto completo de funciones y métodos que permiten adaptarlos adecuadamente a las necesidades del programador.

Medidores

El tipo de aplicaciones {`meter`} permite mostrar dinámicamente valores enteros y en punto flotante mediante una representación gráfica: una aguja en un dial, la posición en una escala rectangular, una pantalla digital estilo calculadora, etc. Las funciones utilizadas para su creación son:

- `standardverticalmeter` y `standardhorizontalmeter` para crear medidores en forma de barra vertical y horizontal, respectivamente.
- `standardroundmeter` para crear medidores con amplitud circular expresada en radianes
- `horizontalmeter` crea un medidor horizontal totalmente personalizable.
- `verticalmeter`, como la anterior pero en vertical.

- `standardperiodicmeter` crea un medidor periódico circular con un periodo de 2π.
- `standard9/3meter` crea un medidor semiesférico horizontal superior.
- `reverse9/3meter` crea uno como el anterior, pero la imagen invertida verticalmente.
- `standard6/12meter` crea un medidor semiesférico vertical izquierdo.
- `standard12/6meter` crea un medidor semiesférico vertical derecho.
- `roundmeter` permite crear un medidor circular personalizable.

Existe un amplio conjunto de métodos para definir cosas tales como la amplitud de la escala, las graduaciones, la imagen y el cursor a utilizar.

Mediante la función `digicounter` es posible crear una aplicación con aspecto similar al de las pantallas de las calculadoras de bolsillo que permite mostrar valores enteros modificables mediante clicks del ratón sobre los distintos dígitos.

Protectores

Durante la fase de desarrollo y depuración de programas es aconsejable utilizar protectores para evitar los errores producidos por el manejo de las imágenes en pantalla. Para ello se utiliza la función `protector`. Cuando una imagen está protegida, los errores son capturados por la función `catcherror` de LE-LISP, mientras que todos los eventos son pasados a la imagen.

B.2.5 Recursos gráficos

En AÏDA, cada objeto gráfico de tipo `application` tiene asociado una ventana LE-LISP cuyo contexto gráfico se construye cuando el objeto es activado en pantalla, por tanto el valor de los atributos gráficos que constituyen el nuevo contexto se toman de los valores actuales en ese momento. Como consecuencia, para una adecuada representación de las aplicaciones, es necesario que el contexto de creación de un objeto sea idéntico al de activación en pantalla de dicho objeto, ya que de lo contrario se producirán efectos no deseados como por ejemplo, que el título de un botón no sea totalmente visible ya que el entorno en el que se creó, según el cual se calcularon sus dimensiones, tenía una fuente más pequeña que la del entorno utilizado cuando se visualizó en pantalla.

La definición de recursos proporciona un método de tratar con entornos gráficos heterogéneos. Mediante la macro `defllresource` se puede definir un nuevo recurso, que puede ser recuperado mediante `llresource`. Aunque el tipo de un recurso puede ser cualquier símbolo arbitrario, los siguientes tipos están predefinidos y poseen un valor por defecto: `foreground`, `background`, `font`, `largefont`, `smallfont`, `attributefont`, `bitmap`, `icon` y `cursor`. Se puede especificar un valor por defecto para cualquier recurso definiendo la función `#:llresource:<tipo-de-recurso>:default`.

Los recursos gráficos definidos por `defllresource` se almacenan en el entorno gráfico actual. Si se desea definir un recurso independiente del entorno, debe utilizarse `defllresource-deferred`.

Se pueden asociar recursos específicamente a una instancia de una aplicación mediante `{application}:resource-name`.

Looks

AÏDA soporta varios *looks*. Mediante la función `current-look` se puede hacer que una aplicación aparezca en pantalla con uno de los looks. Actualmente se soportan los siguientes looks:

- *simple look*, que se corresponde con el utilizado en versiones de AÏDA anteriores a la 1.5.
- ILOG LOOK, que se corresponde con el nuevo look exclusivo de Ilog.
- Motif.
- Open look.

B.2.6 Eventos

El manejo de los eventos relacionados con el VBD está al cargo del programa que se conoce con el nombre de *motor de AÏDA*[36]. Los eventos generados por el VBD son tratados de modo asíncrono, es decir, se procesan solamente cuando el motor está activo, esto es, mientras la función `process-pending-events` se está ejecutando. Cada evento se procesa en cinco fases:

1. **Lectura del evento**.
2. **Búsqueda de la aplicación asociada al evento**. Si la ventana en que se produjo el evento pertenece a AÏDA, la aplicación se encuentra en el campo `appli` de la ventana. Si no, sólo se procesan los eventos **modify**, **kill** y **keyboard-focus**.
3. **Decodificación del evento**. Los *eventos concretos* recibidos en la cola de eventos del VBD se decodifican en *eventos abstractos*. Por ejemplo, cuando una aplicación selecionable recibe un evento concreto como **down-event** con un campo de detalle con valor 0 (botón izquierdo del ratón pulsado), se transforma en un evento abstracto **select** que se interpreta como una petición del usuario para seleccionar la aplicación.
4. **Ejecución del comportamiento de instancia**. El evento original es restaurado una vez que ha sido devuelto el control por parte de la función que define el comportamiento de instancia, con lo cual se evitan posibles perturbaciones debidas a la manipulación del evento por dicha función.
5. **Envío de un mensaje a la aplicación**. Se envía un mensaje con el mismo nombre que el código del evento, con el propio evento como argumento.

Una aplicación puede tomar el control de la cola de eventos por medio de llamadas a las funciones que dan acceso a ella.

[36] AÏDA engine.

B.2.7 Mecanismos de transferencia

AÏDA soporta mecanismos internos de *cortar y pegar* mediante los cuales es posible transferir información de una aplicación a otra de modo instantáneo e interactivo. Para ello se hace de la función `current-selection-client` y de los eventos `generate-copy-selection`, `generate-cut-selection` y `available-selection`.

También soporta mecanismos externos de *cortar y pegar* mediante los cuales se puede transferir texto bidireccionalmente entre AÏDA y aplicaciones externas. Las funciones `display-store-selection` y `display-get-selection` se utilizan para este fin.

También soporta mecanismos de *arrastar y soltar*, aunque sólo entre aplicaciones AÏDA: el objeto se selecciona en una aplicación (denominada *giver*), se elige el destino (una aplicación que recibe el nombre de *catcher*) y se realiza la transferencia. Se dice que la aplicación *giver* ofrece un servicio y que la *catcher* lo acepta. Los pasos involucrados en una operación arrastrar y pegar son:

1. Comienzo de la operación.
2. Preparar al *giver* con `prepare` y destacarlo con `high`.
3. Utilizar un fantasma que sigue el movimiento del ratón. Para se usan las funciones `draw` y `erase`.
4. Búsqueda del *catcher*. Aquellos que acepten servicios del *giver* deben destacarse cuando el cursor del ratón pase sobre ellos. Para ello se utiliza `high` y `unhigh`.
5. El objeto se suelta en un *catcher* que acepta el servicio. Se utilizan las funciones `begin` y `end` para mostrar gráficamente que se ha aceptado el servicio.
6. Si el objeto se suelta en un *catcher* que no acepta el servicio o sobre un objeto que no es un *catcher*, se informa al *giver* del fallo de la operación por medio de `refused`.
7. El *giver* retorna a su estado original por medio de `unhigh`.

B.2.8 Herramientas

AÏDA proporiciona un conjunto de herramientas que facilitan el desarrollo de programas. Comprenden:

- **Ayuda on-line**. Muestra la documentación y un ejemplo de cada imagen y aplicación predefinida en AÏDA. El código fuente de los ejemplos es accesible y puede ser modificado libremente. El código modificado se puede evaluar desde la misma ayuda, lo que permite observar inmediatamente los cambios. También se muestra la jerarquía de tipos así como los campos y métodos asociados a cada tipo. Desde la ayuda también se tiene acceso directo a las herramientas de desarrollo.
- **Editor de ficheros**. Permite editar cómodamente los ficheros que contienen los programas AÏDA. Contiene un editor especialmente diseñado para editar funciones en LISP. Se puede personalizar.
- **Editor de objetos**. Permite editar objetos mediante el uso del ratón. Posee opciones de *zoom* y *unzoom* que permiten editar objetos que a su vez forman parte de otros objetos.

- **Visor de la estructura de imágenes**. Permiten mostrar en forma de árbol aquellas partes de la jerarquía de imágenes que se desea observar.
- **Apropos**. Es una herramienta de búsqueda que acepta cualquier nombre y devuelve todas las entidades AÏDA que contiene dicho nombre. Para cada entidad, proporciona su tipo y la página del manual LE-LISP o AÏDA en la que se trata.
- **Editor de menús**. Es una herramienta que permite crear y editar interactivamente menús de AÏDA que posteriormente podrán ser integrados en cualquier programa.
- **Editor de iconos**. Incluye un editor punto a punto adecuado para la edición de bitmaps no excesivamente grandes.
- **Aïdapaint**. Es una herramienta de dibujo de bitmaps bastante completa, con una amplia barra de herramientas y múltiples opciones de edición.
- **Editor de colores**. Muestra un una rueda con toda la gama de colores disponible. En un display digital se muestran los componentes RGB del punto sobre el que está situado el ratón. A la inversa, el usuario puede especificar los valores RGB y el puntero del ratón se posicionará sobre el color correspondiente.
- **Editor de árboles**. Consta básicamente de una aplicación de tipo editor de árboles y de un menú. El menú se muestra cuando el usuario pulsa el ratón en el fondo del editor.
- **Calculadora**. Permite realizar operaciones matemáticas comunes.

Apéndice C

El componente gráfico de Centaur

CENTAUR es una herramienta orientada al diseño de lenguajes de programación que ha sido desarrollada en el INRIA[1] de Sophia-Antipolis (Francia). CENTAUR ha sido construido a partir de LE-LISP, Mu-Prolog, VTP[2] y X Window System. Una de sus características más importantes en lo que respecta a la construcción de aplicaciones gráficas es que incorpora parte del lenguaje de imágenes de AÏDA y un cierto número de aplicaciones equivalentes a parte de las disponibles en AÏDA.

C.1 Componentes del sistema CENTAUR

Además de incorporar completamente los lenguajes LE-LISP y Mu-Prolog y un sistema de objetos gráficos, CENTAUR proporciona los siguientes entornos de desarrollo:

- El entorno METAL para la edición, revisión y compilación de programas METAL, un metalenguaje para la definición de sintaxis abstractas y concretas. El compilador de METAL utiliza Lex y Yacc para generar los analizadores y VTP para construir los árboles de sintaxis abstracta.

- El entorno PPML para editar y compilar programas PPML[3], un metalenguaje utilizado para transformar árboles de sintaxis abstracta en una representación textual fácilmente comprensible por el usuario.

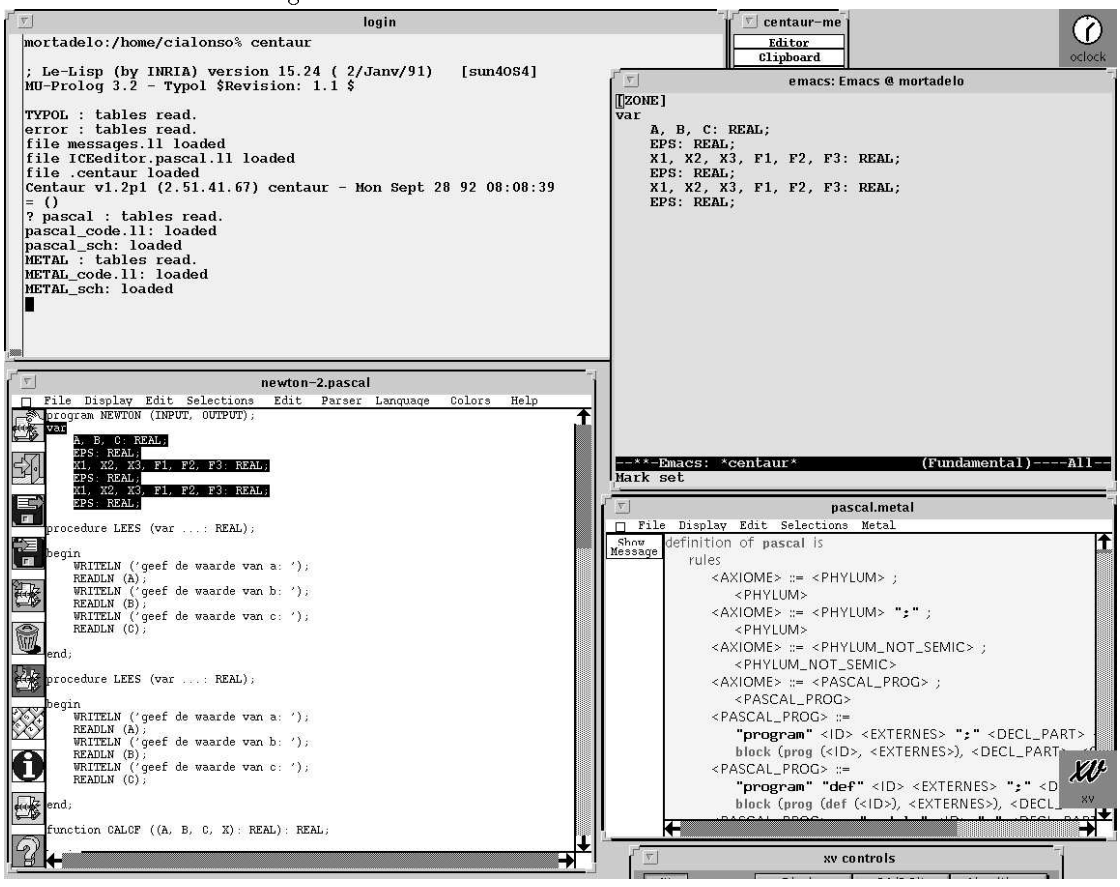
- El entorno TYPOL para editar y depurar especificaciones TYPOL y para facilitar la comprobación de tipos y la ejecución de programas escritos en TYPOL, un lenguaje de especificación basado en reglas de inferencia que permite definir tanto la semántica estática como la dinámica de un lenguaje de programación.

- El sistema ASF+SDF para la manipulación interactiva de especificaciones escritas en ASF+SDF, que combina el formalismo de definición sintáctica SDF[4] con el formalismo ASF[5] para la especificación de tipos de datos abstractos.

[1] **I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique.
[2] **V**irtual **T**ree **P**rocessor, un sistema para la manipulación de árboles de sintaxis abstracta.
[3] **P**retty **P**rinter **M**eta **L**anguage.
[4] **S**yntax **D**efinition **F**ormalism.
[5] **A**bstract **S**pecification **F**ormalism.

- El editor CENTAUR, que permite la edición estructurada utilizando la especificación abstracta de un lenguaje que puede estar escrita en METAL o SDF.
- GSE[6], una herramienta para la generación de editores dirigidos por la sintaxis utilizada en ASF+SDF.
- FIGUE[7], diseñado para trabajar con árboles de objetos gráficos anidados.
- GRAPH, un servidor gráfico que permite a las aplicaciones clientes mostrar en pantalla estructuras gráficas complejas. El protocolo de comunicación entre el servidor y el cliente establece el método en que dichos elementos colaborarán para permitir la edición y manipulación de dichas estructuras gráficas.

Figura C.1: El entorno de desarrollo de CENTAUR



En la figura C.1 se muestra un entorno de trabajo típico de Centaur, en el que se encuentran abiertos dos editores, uno con la especificación METAL del lenguaje Pascal y otro conteniendo un lenguaje escrito en ese mismo Pascal. Un bloque de éste último programa se encuentra seleccionado para poder ser editado en el editor Emacs enlazado a CENTAUR.

[6] **G**eneric **S**yntax-directed **E**ditor.
[7] **F**ormatage **I**ncremental **G**raphiq**UE**.

Para una descripción general de los distintos componentes, puede consultarse [Centaur 92a]. Para una descripción más detallada debe consultarse [Centaur 92b]. Si se desean detalles de la implementación interna, entonces lo adecuado es recurrir a [Centaur 92c]. Los componentes FIGUE y GRAPH no aparecen referenciados en la documentación general del sistema CENTAUR. Sin embargo, en los directorios `contrib/figue/docs` y `contrib/graphics/docs` de la cinta de distribución de CENTAUR se encuentran disponibles ficheros en formato ASCII y PostScript con la documentación de ambos componentes.

C.2 Los objetos gráficos de CENTAUR

Los objetos gráficos se agrupan en lo que se denomina el sistema *gfxobj*, creado a partir de la jerarquía de constructores de imágenes de AÏDA. Sin embargo, una diferencia inportante reside en el hecho de que CENTAUR no incorpora las aplicaciones AÏDA, sino que dispone de su propia jeraquía de aplicaciones.

La raiz de la jerarquía de los objetos propios de *gfxobj* se ha establecido en {`object`}[8]. Directamente bajo esta clase se encuentra {`gfxobj`}, que es la superclase de todos los tipos de objetos gráficos de *gfxobj*.

Para una completa descripción del componente gráfico de CENTAUR es conveniente consultar [Centaur 92c].

C.2.1 Los constructores de imágenes

Además de los constructores de AÏDA `rectangle`, `icon`[9], `box`, `translation`, `centaeredimage`, `view`, `row` y `column`, CENTAUR incorpora loas siguientes objetos que ignoran los mensajes de cambio de tamaño: `fixrectangle`, `fixrow` y `fixcolumn`.

C.2.2 Los objetos gráficos básicos de *gfxobj*

A continuación se dará una lista con los objetos gráficos disponibles bajo la jerarquía de {`gfxobj`}. Pero antes es conveniente realizar una pequeña explicación sobre el modo en que dichos objetos se muestran en pantalla, ya que presenta ligeras diferencias con respecto a AÏDA.

Al igual que en AÏDA, existe una función `add-application` para inicializar un objeto gráfico, pero con la diferencia de que en CENTAUR está función no hace visible el objeto en pantalla. Esto conlleva que la creación y visulización de un objeto gráfico en CENTAUR sea un proceso de tres fases:

1. Crear el objeto usando la función correspondiente.
2. Realizar una llamada a `add-application` con el objeto como argumento.
3. Enviar al objeto el mensaje `show` mediante la función `send`.

[8] Equivalente a {**`application`**} en AÏDA.

[9] En CENTAUR se utiliza la función **`gfxobj-load-icon`** en vez de **`libloadicon`** para crear un icono a partir de un fichero bitmap almacenado en disco.

Posteriormente se pueden utilizar envios sucesivos de los mensajes `hide` y `show` para controlar la desaparición y reaparición del objeto en pantalla. También se dispone de los métodos {`gfxobj`}`:display` y {`gfxobj`}`:redisplay`, equivalente a los mensajes `display` y `redisplay` de AÏDA.

En CENTAUR no está disponible la función `remove-application`, por lo que la eliminación de un objeto gráfico se realiza enviándole el mensaje `terminate`.

Botones

Existen dos tipos de botones diferentes:

- *Push buttons*, creados mediante {`butobj`}`:create`, que se corresponden con los botones estándar de AÏDA.
- *Trill buttons*, creados mediante {`trill`}`:create`, que son botones de autorepetición en los cuales la función asociada al botón se está disparando continuamente mientras el botón del ratón esté pulsado dentro del *trill button*.

Enviando el mensaje `action` con un botón y una función como argumentos se puede asociar dicha función como la acción a realizar por el botón cuando éste se encuentre pulsado.

Menús

Centaur proporciona tres tipos de menús:

- Menús fijos, siempre presentes en la pantalla, que se crean mediante llamadas a {`menu`}`:create`.
- Menús *pulldown*, que se crean mediante llamadas a {`pulldown`}`:create`.
- Barras de menús, una fila de menús pulldown, creados mediante {`menubar`}`:create`.

Se puede asociar a cualquier objeto de tipo {`gfxong`} un menú *popup*[10] que aparecerá cuando se pulse el botón derecho del ratón sobre dicho objeto. Para ello es necesario seguir los siguientes pasos:

1. Crear un menú de tipo {`menu`}.
2. Llamar a la primitiva {`gfxobj`}`:add-popup` con el tipo de objeto al que se quiere asociar el menú como argumento.
3. Utilizar {`gfxobj`}`:set-popup` para asociar el menú a una instancia del tipo de objeto referido en el paso enterior.

Terminales virtuales

Se crean mendiante llamadas a la función `tty-application`.

[10] Los menús *popup* también se denominan *menús de aparición súbita.*

Scrollers

Permiten desplazar un objeto dentro de un área de visión más pequeña que el tamaño del propio objeto. Se crean mediante {scrollobj}:create. Utilizando {scrollobj}:add-acrollbar y {srollobj}:delete-scrollbar se pueden añadir o eliminar del scroller las barras de desplazamiento vertical u horizontal.

Existe un conjunto de métodos para establecer la posición de las barras de scroll y su tamaño. Otro conjunto de métodos permite desplazar el objeto contenido en el scroller.

Mediante la redefinicón del mensaje scrool-resize se le puede indicar a un scroller cómo debe comportarse ante la presencia de eventos de redimensionamiento[11].

C.2.3 Creación de nuevos *gfxobj*

Cuando se crea un nuevo objeto gráfico dentro de la jerarquía {gfxobj}, como mínimo se debe crear un método create para dicho objeto[12] que deberá realizar las siguientes tareas:

- Utilizar el mensaje new para crear un nueva instancia del objeto.
- Inicializar los campos de esa nueva instancia utilizando los valores pasados como argumentos a create.
- Almacenar en el campo heredado image la imagen del objeto.

Para establecer la posición, tamaño, imagen y ventana de un objeto gráfico se utiliza la función make-gfxobj. Si el primer argumento de esta función es un símbolo, entonces crea un nuevo objeto del tipo indicado por el símbolo. En otro caso utiliza el objeto pasado como primer argumeto para asignarle la posición, tamaño e imagen indicada en los siguientes argumentos.

C.2.4 La gestión de eventos en *gfxobj*

La gestión de los eventos para los objetos gráficos de *gfxobj* se realiza de manera similar a como se hacía en AÏDA. Dado un evento que se quiere tratar de modo especial en un objeto, se define un método para ese objeto que tiene por nombre el nombre de evento y que tomará como argumento una instancia del objeto y una instancia de {event}. Se pueden redefinir tanto los eventos abstractos como los concretos.

El comportamiento de los objetos básicos en *gfxobj* se definió utilizando Esterel, un lenguaje de programación especialmente adaptado para trabajar con sistemas reactivos.

C.3 El editor ctedit

El editor ctedit permite visualizar y modificar datos estructurados de CENTAUR. Los datos mostrados en un ctedit son formateados según un *pretty printer*. La clase {ctedit} es una subclase de {gfxobj}.

[11] Existe un método **scroll-resize** genérico en la clase {**gfxobj**} que no hace nada, de ahí la necesidad de redefinirlo.

[12] A menos que el nuevo objeto que se desee crear no sea más que un subobjeto de alguno ya existente al que tan sólo se desea modificar su comportamiento mediante la redefinición de alguos métodos como **redisplay** o **grow**.

Para crear un editor de este tipo se usa {ctedit}:create. Para obtener el *sujeto*[13] que va a ser editado, se utiliza {ctedit}:subject. Mediante {ctedit}:get-formatter se obtiene el formateador utilizado para editar un determinado *sujeto*.

C.3.1 El formateador

El componente principal de un ctedit es el formateador, que almacena tanto el objeto que va a ser editado como la información del *pretty printer*. Para definir el formateador se utiliza el lenguaje PPML. El nombre simbólico que recibe el formateador es buffer.

Para actualizar los contenidos del editor de acuerdo con el formateador se utilizan los métodos {ctedit}:redraw y {ctedit}:incremental-update. El primero recalcula totalmente los datos del editor mientras que el segundo realiza una actualización incremental sobre los cambios realizados. Ninguno de los dos métodos revisualiza los contenidos del editor.

C.3.2 La visualización del contenido del editor

Existen tres modos diferentes en que puede actuar una instancia de ctedit para visualizar los datos:

- En el modo *expose*, el servidor X borra las regiones modificadas y genera eventos Expose para esas regiones. La ventana del ctedit recibe esos eventos y redibuja los objetos involucrados. Es el modo por defecto.
- En el modo *refresh*[14] el ctedit envía una petición clear-window[15] al servidor X. El servidor borra la ventana y envía eventos Expose. Dichos eventos son recibidos por el ctedit mientras se encuentre en modo *refresh*.
- El modo *update* es parecido al modo *refresh*, salvo que la petición de borrado de la ventana al servidor es sustituida por una petición *expose-window*, por lo que el servidor no destruye los contenidos de la ventana sino que genera eventos Expose para todas sus regiones visibles.

El método {ctedit}:incremental-redisplay cambia el modo del ctedit a *update*. Una vez que el servidor X genera todos los eventos Expose, restaura el ctedit al modo *expose*.

Mediante {ctedit}:expose se puede redibujar completamente el ctedit, puesto que este método establece el modo a *user-expose* para posteriomente restaurarlo a *expose*.

C.3.3 Los eventos del ratón

Cada ctedit tiene asociado una *mouseEventList*, una lista de pares <botón> . <tipo>. Puede haber a lo sumo tres pares, correspondientes a los tres botones del ratón. Si no hay un par para un botón, entonces ese botón no realiza ninguna acción.

El valor de <botón> puede ser left, midle o right. Cada botón definido llama a una de las siguinetes funciones según el tipo de evento:

[13] En la terminología utilizada por los creadores de ctedit, se denomina *sujeto* a los datos almacenados en una instancia de ctedit.

[14] También llamado *user-expose*.

[15] Una llamada a la función XClearWindow de la librería Xlib.

- <tipo>:mouse-down
- <tipo>:mouse-drag
- <tipo>:mouse-up

Cada una de esas funciones toma como argumento una instancia de `ctedit` y una instancia de `event`.

C.3.4 Selecciones

Se pueden asociar selecciones a un `ctedit`. Las selecciones se encapsulan en *selects*, cuya implementación depende del formateador, que son las que almacenan los atributos de la representación gráfica de los objetos.

Cuando se utiliza PPML, se puede crear una *select* asociada al ratón mediante la siguiente llamada[16]:

```
({ctedit}:create-select <ctedit> '{current} '{textual}:all)
```

Existen métodos asociados a `ctedit` que permiten crear, añadir, encontrar y borrar *selects* y selecciones. También hay dos métodos `update-select` y `update-selection` que permiten recalcular incrementalmente la imagen del `ctedit`. Estos dos métodos recalculan la imagen, pero no la redibujan. Para ello es necesario llamar a `incremental-redisplay`.

[16] El tipo de *select* `'{textual}:all` destaca todas las palabras de una expresión. Otro tipo de *select* es `'{textual}:immediate`, que destaca sólo aquellos tokens impresos por una regla.

Bibliografía

[Aho et al. 90] Aho, A. V.; Sethi, R. y Ullman, J. D.
Compiladores. Principios, Técnicas y Herramientas.
1990. Addison-Wesley Iberoamericana S.A., Wilmington, Delaware, U.S.A.

[Aho y Ullman 73] Aho, A. V. y Ullman, J. D.
The Theory of Parsing, Translation and Compiling, vol. 1 y 2.
1973. Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A.

[Billot y Lang 89] Billot, S. y Lang, B.
The Structure of Shared Forest in Ambiguous Parsing.
1989. Research Report 1038, INRIA Roquencourt, Francia.

[Centaur 92a] *Centaur 1.2: The Crust.*
1992. Centaur Distribution, INRIA Sophia-Antipolis, Valbonne, Francia.

[Centaur 92b] *Centaur 1.2: The Mantle.*
1992. Centaur Distribution, INRIA Sophia-Antipolis, Valbonne, Francia.

[Centaur 92c] *Centaur 1.2: The Core.*
1992. Centaur Distribution, INRIA Sophia-Antipolis, Valbonne, Francia.

[Coffin 89] Coffin, S.
UNIX. Manual de Referencia.
1989. Mc Graw-Hill Interamericana de España, Madrid, España.

[Donnelly y Stallman 88] Donnelly, C. y Stallman, R.
BISON. The YACC-compatible Parser Generator.
1988. Free Software Foundation, Cambridge, Massachussets, U.S.A.

[Earley 70] Earley, J.
An Efficient Context-Free Parsing Algorithm.
1970. Communications of the ACM, vol. 13,2 (pp. 94–102).

[Gulías 94] Gulías Fernández, V. M.
Interfaz gráfica para una implementación del λ-cálculo, construida sobre entornos heterogéneos utilizando una arquitectura cliente-servidor.
1994. Tesina de Licenciatura. Facultade de Informática, Universidade da Coruña, La Coruña, España.

[Harrison 87] Harrison, M. A.
Introduction to Formal Language Theory.
1987. Addison-Wesley, Reading, Massachusetts, U.S.A.

[Hopcroft y Ullman 79] Hopcroft, J. E. y Ullman, J. D.
Introduction to Authomata Theory, Languages and Computation.
1979. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, U.S.A:

[Hopgood et al. 83] Hopgood, Duce, Gallap y Sutcliffe.
Introduction to the Graphics Kernel System (GKS).
1983. Academic Press, U.S.A.

[IBM 92] *IBM AIXwindows Programming Guide.*
1992. IBM International Technical Support Center, Austin, Texas, U.S.A.

[ILOG 91a] *MASAÏ Version 1.35 Dialog Manager.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 91b] *MASAÏ Version 1.35 User's Guide.*
1991. ILOG S.A., Gentilly, Francia.

[ILOG 91c] *PostScriptLink Version 1.25. Reference Manual.*
1991. ILOG S.A., Gentilly, Francia.

[ILOG 92a] *AÏDA Version 1.65 Extensions.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92b] *AÏDA Version 1.65 Programming Guide.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92c] *AÏDA Version 1.65 Reference Manual.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92d] *AÏDA Version 1.65 Release Notes and Addendum.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92e] *AÏDA Version 1.65 User's Manual.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92f] *LE-LISP-AÏDA Runtime Generator for Unix Version 1.65.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92g] *MASAÏ Version 1.35 Reference Manual.*
1992. ILOG S.A., Gentilly, Francia.

[ILOG 92h] *MASAÏ Version 1.35 Release Notes and Addenda.*
1992. ILOG S.A., Gentilly, Francia.

[INRIA 91] *LE-LISP Version 15.25 Reference Manual.*
1991. INRIA Rocquencourt, Le Chesnay, Francia.

[Johnson 75] Johnson, S.C.
YACC: Yet Another Compiler Compiler.
1975. Computer Sciences Technical Report n. 32, AT&T Bell Laboratories, Murray Hill, New Jersey, U.S.A.

[Johnson y Reichard 92] Johnson, E.F. y Reichard, K.
Programación y Aplicaciones X Window.
1992. Ra-ma, Madrid, España.

[Kerninghan y Ritchie 85] Kerninghan, B. W. y Ritchie, D.
El Lenguaje de Programación C.
1985. Addison-Wesley Iberoamericana, S.A., Wilmington, Delaware, U.S.A.

[Kerninghan y Pike 83] Kerninghan, B. W. y Pike, R.
The Unix Programming Environment.
1983. Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A.

[Korth y Silberschatz 87] Korth, H. F. y Silberschatz, A.
Fundamentos de Bases de Datos.
1987. Libros McGraw-Hill de México, S.A. de C.V., Naucalpan de Juárez, Méjico.

[Krulee 91] Krulee, G. K.
Computer Processing of Natural Languages.
1991. Prentice-Hall, Englewood Cliffs, New Jersey, U.S.A.

[Lewis et al. 76] Lewis, P. H.; Rosenkrantz, D. J. y Stearns, R. E.
Compiler Design, Theory.
1976. Addison-Wesley, Reading, Massachusetts, U.S.A.

[Mason y Brown 90] Mason, T. y Brown, D.
LEX & YACC.
1990. O'Reilly & Associates, Inc., Sebastopol, California, U.S.A.

[Mui y Pearce 92] Mui, L. y Pearce, E.
X Window System Administrator's Guide.
1992. O'Reilly & Associates, Inc., Sebastopol, California, U.S.A.

[Paxon 94] Paxon, V.
Flex 2.4.6 (January 1994). Reference Documentation, Full User Documentation & Changes between Releases.
1994. Lawrence Berkeley Laboratory, University of California. Berkeley, California, U.S.A.

[Quercia y O'Reilly 90] Quercia, V. y O'Reilly, T.
X Window System User's Guide.
1990. O'Reilly & Associates, Inc., Sebastopol, California, U.S.A.

[Reiss y Radlin 93] Reiss, L. y Raddin, J.
Aplique X Window.
1993. McGraw-Hill Interamericana de España S.A., Madrid, España.

[Sánchez y valverde 89] Sánchez Dueñas, G. y Valverde Andreu, J. A.
Compiladores e Intérpretes: un Enfoque Pragmático, segunda edición.
1989. Díaz de Santos, Madrid, España.

[Sanchís y Galán 88] Sanchís Llorca, F. J. y Galán Pascual, C.
Compiladores: Teoría y Construcción, segunda edición.
1988. Ed. Paraninfo, Madrid, España.

[Sudkamp 88] Sudkamp, T. A.
Languages and Machines. An Introduction to the Theory of Computer Science.
1988. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, U.S.A.

[SunSoft 93] *Lex and Yacc* in *Sun OS 5.2 Programming Utilities.*
1993. SunSoft Inc., Mountain View, California, U.S.A.

[Vilares 89] Vilares Ferro, M.
An Incremental Approach for the Earley's Algorithm.
1989. CERICS Internal Report, Sophia-Antipolis, Valbonne, Francia.

[Vilares 92] Vilares Ferro, M.
Efficient Incremental Parsing for Context-Free Languages.
1992. Tesis Doctoral, Université de Nice, NIza, Francia.

[Vilares 93] Vilares Ferro, M.
An Efficient Context Free Backbone for Natural Languages Analyzers.
1993. Proc. of SEPLN'93, Santiago de Compostela, La Coruña, España.

[Vilares y Dion 94] Vilares Ferro, M. y Dion, B.
Efficient Incremental Parsing for Context-Free Languages.
1994. Proc. of the Fifhty IEEE International Conference on Computer Languages, Toulouse, Francia.

[Vilares y Graña 93] Vilares Ferro, M. y Graña Gil, J.
Parsing as Resolution.
1993. Proc. of the First Compulog Network Meeting of Parallelism and Implementation Technologies, Madrid, España.

[Villemonte de la Clergerie 90] Villemonte de la Clergerie, E.
DyALog. Une Implementation des Clauses de Horn en Programmation Dynamique.
1990. Actes du 9^{th} Séminaire de Programmation Logique, CNET, Lannion, Francia (pp. 207–228).

[Winston y Horn 91] Winston, H. W. y Horn, B.
LISP, tercera edición.
1991. Addison-Wesley Iberoamericana S.A., Wilmington, Delaware, U.S.A.