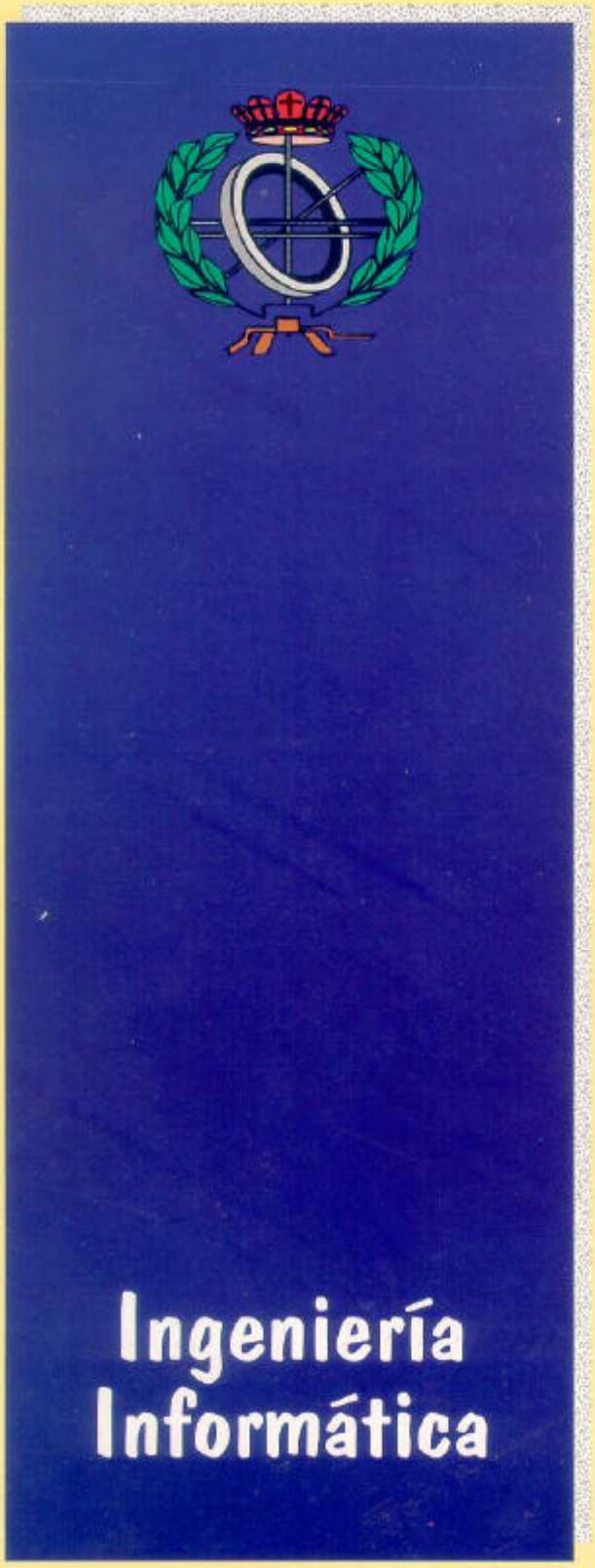
Cuadernos Didácticos



Conceptos básicos de Procesadores de Lenguaje

Cuaderno N° 10

Juan Manuel Cueva Lovelle
Área de Lenguajes y Sistemas Informáticos
Departamento de Informática
Universidad de Oviedo

Oviedo, Diciembre 1998

Cuadernos Didácticos
Ingeniería Informática

Cuaderno Nº 10

Conceptos básicos de Procesadores de Lenguaje

Autor:

J.M. Cueva Lovelle
Dpto. de Informática
Universidad de Oviedo - España

Editorial:
SERVITEC

ISBN: 84-8416-889-1
Deposito Legal: AS-3552-98

Oviedo, Diciembre 1998

1ª Edición

Consultor Editorial
Juan Manuel Cueva Lovelle
cueva@lsi.uniovi.es

CONCEPTOS BÁSICOS

DE PROCESADORES DE LENGUAJE

Juan Manuel Cueva Lovelle

Catedrático de E.U. de Lenguajes y Sistemas Informáticos
Departamento de Informática
Universidad de Oviedo

Diciembre 1998

PRÓLOGO

El objetivo de este Cuaderno Didáctico es enseñar al alumno los problemas y técnicas que se plantean en el diseño y construcción de procesadores de lenguaje. El estudio de estas técnicas permite al alumno una visión más amplia de los lenguajes de programación, habitualmente estudiados desde el punto de vista del programador y no de las interioridades inherentes a su diseño e implementación.

La utilidad de profundizar en los procesadores de lenguaje se puede resumir en los siguientes puntos:

- *Conocer mejor el lenguaje que se utiliza habitualmente.*
- *Ampliar los conocimientos sobre la implementación interna de las estructuras de datos y clases de los lenguajes de programación.*
- *Posibilidad de comparar entre distintos lenguajes de programación, mejorando la capacidad de selección de un lenguaje.*
- *Estudiar las relaciones entre los procesadores de lenguaje y la arquitectura de los ordenadores.*
- *Facilitar la evaluación y elección de las herramientas de manejo de lenguajes (compiladores, intérpretes, etc...)*
- *Aprender a diseñar lenguajes de programación.*

En primer lugar se realiza una panorámica general de los lenguajes de programación y otros tipos de lenguajes presentes actualmente en Informática. A continuación se definen los procesadores de lenguaje y se especifican algunos de sus tipos. El tercer apartado está dedicado a las arquitecturas de ordenadores y su influencia en la construcción de procesadores de lenguaje. A continuación se incluyen varios epígrafes sobre portabilidad y puesta en marcha de compiladores.

En el apartado 7 comienza un conjunto de apartados sobre la especificación de lenguajes.

A partir del apartado 12 se van describiendo los distintos módulos que conducen a la construcción de traductores, compiladores, e intérpretes introduciendo los conceptos necesarios para su comprensión.

La metodología de desarrollo de un procesador de lenguaje es un caso particular de la ingeniería del software. La fase de análisis está constituida por la definición de los requisitos del lenguaje fuente y el lenguaje objeto si lo hubiera, por medio de su especificación léxica, sintáctica y semántica. El segundo paso es el diseño preliminar donde se defienen las características de los módulos que van a componer el sistema, es decir los analizadores: léxico, sintáctico y semántico, la tabla de símbolos, el tratamiento de errores, y la generación de código intermedio. Otros módulos posibles son el generador de código objeto, el optimizador de código, el intérprete. El siguiente paso es el diseño detallado donde se especifican las operaciones que se realizarán en cada módulo. El último paso es la implementación, en este caso cada módulo se implementa como una clase y las operaciones externas sobre la clase serán los métodos públicos y los datos y métodos relativos a la implementación constituirán la parte privada. El resultado final es el procesador de lenguaje como un módulo que utiliza los objetos que son instancias de las distintas clases.

También se incluyen unas breves descripciones de las herramientas de desarrollo de procesadores de lenguaje. Por último se incluye una reseña histórica de los procesadores de lenguaje y una bibliografía comentada.

Las descripciones se complementan con un conjunto de ejemplos, que culminan en el desarrollo completo de pequeños compiladores, traductores e intérpretes. También se incorporan ejercicios propuestos con distintos grados de dificultad. El desarrollo de los procesadores de lenguaje se realiza en el lenguaje C++, aprovechando sus características de orientación a objetos, diseño modular y eficiencia.

CONTENIDOS

1. LENGUAJES DE PROGRAMACIÓN

Las relaciones humanas se llevan a cabo a través del lenguaje. Una lengua permite la expresión de ideas y de razonamientos, y sin ella la comunicación sería imposible. Los ordenadores sólo aceptan y comprenden un lenguaje de bajo nivel, que consiste en largas secuencias de ceros y unos. Estas secuencias son ininteligibles para muchas personas, y además son específicas para cada ordenador, constituyendo el denominado *lenguaje máquina*.

La programación de ordenadores se realiza en los llamados *lenguajes de programación* que posibilitan la comunicación de órdenes a la computadora u ordenador.

Un **lenguaje de programación** se puede definir de distintas formas:

- *es una notación formal para describir algoritmos o funciones que serán ejecutadas por un ordenador*.
- *es un lenguaje para de comunicar instrucciones al ordenador*.
- *es una convención para escribir descripciones que puedan ser evaluadas.*

También se utilizan en Informática *otros lenguajes que no son de programación* y que tienen otras aplicaciones, por ejemplo puede haber lenguajes para describir formatos gráficos, de texto, de descripción de páginas, de descripción de sonidos y música, etc... En otros casos los lenguajes pueden ser un subconjunto de los lenguajes naturales (Castellano, Inglés, etc...) o lenguajes que describen un formato particular de entrada de datos.

1.1 Clasificación de los lenguajes de programación

Los lenguajes de programación se pueden clasificar desde distintos puntos de vista:

- *Según su grado de independencia de la máquina*
- *Según la forma de sus instrucciones*
- *Por generaciones*

• Según su grado de independencia de la máquina

Los lenguajes se pueden clasificar según su grado de independencia de la máquina en que se ejecutan en cinco grupos:

§ *Lenguaje máquina*
§ *Lenguaje ensamblador* (en inglés *assembly*)
§ *Lenguajes de medio nivel*
§ *Lenguajes de alto nivel* o lenguajes orientados a usuarios
§ *Lenguajes orientados a problemas concretos*

§ ***El lenguaje máquina*** es la forma más baja de un lenguaje de programación. Cada instrucción en un programa se representa por un código numérico, y unas direcciones (que son otros códigos numéricos) que se utiliza para referir las asignaciones de memoria del ordenador. El lenguaje máquina es la notación que entiende directamente el ordenador, está en binario o en notación hexadecimal, y el repertorio de sus instrucciones, así como la estructura de éstas, están ligadas directamente a la arquitectura de la máquina. Los ordenadores actuales se basan en la arquitectura de la máquina de *Von Neumann* [BURK46, FERN94]. La máquina de *Von Neumann* tiene un repertorio de instrucciones basado en datos, operaciones aritméticas, asignaciones de posiciones de memoria y control de flujo. Véase en el anexo I una breve descripción del lenguaje máquina de los microprocesadores 80x86.

§ ***El lenguaje ensamblador*** es esencialmente una versión simbólica de un lenguaje máquina. Cada código de operación se indica por un código simbólico. Por ejemplo ADD para adición y MUL para multiplicación. Además, las asignaciones de memoria se dan con nombres simbólicos, tales como PAGO y COBRO. Algunos ensambladores contienen macroinstrucciones cuyo nivel es superior a las instrucciones del ensamblador. Véase el anexo II el lenguaje ensamblador de la familia de los microprocesadores 80x86.

§ ***Los lenguajes de medio nivel*** tienen algunas de las características de los lenguajes de bajo nivel (posibilidad de acceso directo a posiciones de memoria, indicaciones para que ciertas variables se almacenen en los registros del microprocesador, etc...) añadidas a las posibilidades de manejo de estructuras de control y de datos de los lenguajes de alto nivel. Ejemplos de este tipo de lenguajes son el C [KERN78, KERN88] y el FORTH [KELL86]. Sobre el lenguaje c puede consultarse el anexo IV.

§ ***Los lenguajes de alto nivel*** tales como FORTRAN [ANSI89, CUEV93b], COBOL [ANSI74], Pascal [JENS91, CUEV94A],... tienen características superiores a los lenguajes de tipo ensamblador, aunque no tienen algunas posibilidades de acceso directo al sistema. Facilitan la escritura de programas con estructuras de datos complejas, la utilización de bloques, y procedimientos o subrutinas. Sobre el lenguaje Pascal puede consultarse el anexo III. Dentro de los lenguajes de alto nivel se pueden destacar un tipo de lenguajes, denominados *lenguajes orientados a objetos*, que permiten definir tipos abstractos de datos (habitualmente denominados *clases*) que agrupan datos y métodos (operadores, funciones y procedimientos). Los objetos son la instanciación de una clase. Las clases se definen en tiempo de compilación, y los objetos son definidos en ejecución. Las clases pueden heredar propiedades de otras clases (*herencia*). El acceso a los datos de un objeto tan sólo se hace a través de sus métodos (*encapsulación*). Los métodos con un mismo nombre pueden manejar distintos tipos de objetos (*polimorfismo*), detectando el

método en tiempo de ejucución la operación que debe realizar sobre el objeto (asociación dinámica). Ejemplos de este tipo de lenguajes son Smalltalk [GOLD83, GOLD89], C++ [STRO86, STRO91, STRO94], Eiffel [MEYE88, MEYE92], Object Pascal y Turbo Pascal® [CUEV94A], etc... Sobre los lenguajes orientados a objetos pueden consultarse los anexos V (Simula), VI (Smalltalk), VII (Ada), VIII (C++), IX (Object Pascal y Turbo Pascal), X (CLOS), XI (Eiffel) y XX (Java).

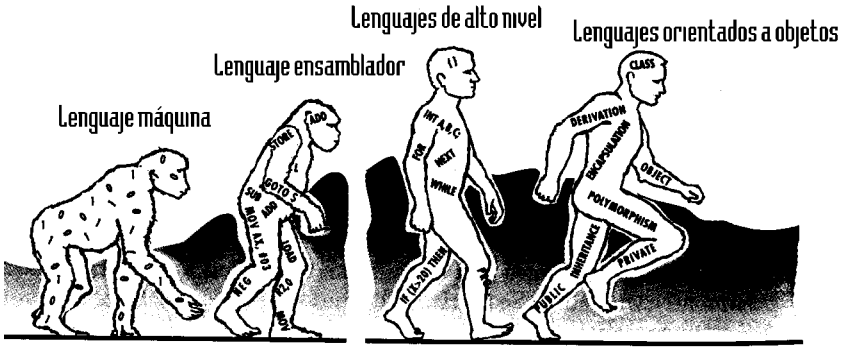


Fig. 1: La evolución de los lenguajes de programación

§ ***Los lenguajes orientados a problemas concretos*** se utilizan para la resolución de problemas en un campo específico. Ejemplos de tales lenguajes son el SQL (ver anexo XIII) y XBASE para el manejo de bases de datos; SPSS® (ver anexo XIV) y BMDP® para cálculos estadísticos; Postscript® (ver anexo XII) y True Page® para la descripción de páginas; y el COGO para aplicaciones en ingeniería civil.

• Según la forma de sus instrucciones

Los lenguajes se pueden clasificar según la forma de sus instrucciones o según los distintos paradigmas de programación:

§ *Lenguajes imperativos o procedimentales*
§ *Lenguajes declarativos: lógicos y funcionales*
§ *Lenguajes concurrentes*
§ *Lenguajes orientados a objetos*

La clasificación anterior no es excluyente entre sí, es decir un lenguaje puede estar incluido en más de un paradigma. Así por ejemplo el lenguaje Ada, es un lenguaje imperativo, tiene mecanismos de encapsulación y genericidad propios de los lenguajes basados en objetos, y permite la concurrencia.

§ ***Los lenguajes imperativos o procedimentales*** son los que usan la instrucción o sentencia de asignación como construcción básica en la estructura de los programas. Son lenguajes orientados a instrucciones, es decir la unidad de trabajo básica de estos lenguajes es la instrucción o sentencia. Ejemplos de lenguajes imperativos son: Pascal [JENS91, CUEV94A], C [KERN78, KERN88], C++ [STRO86, STRO91, STRO94], Ada [ANSI83], FORTRAN [ANSI89, CUEV93b], COBOL [ANSI74], Modula-2 [WIRT88], Oberon [REIS92a, REIS92b, WIRT92], etc...

Históricamente son los primeros lenguajes, y actualmente son los más usados. FORTRAN, COBOL y ALGOL son los precursores y su influencia se ha propagado hasta la actualidad (fig. 2). *Backus* expresó que el diseño de estos lenguajes estaba influido en gran parte por la máquina de *Von Neumann* [BURK46, FERN94], dado que una CPU acoplada a una memoria y a unos canales de entrada salida, implicaba un tipo de lenguajes cuyas características pretendían sacar el máximo partido a este tipo de arquitectura. Así este tipo de lenguajes se caracteriza por:

¤ *Uso intensivo de variables*. Uno de los componentes principales de la arquitectura de *Von Neumann* es la memoria, una forma de referenciar las posiciones de memoria de los lenguajes de alto nivel es asociarles un identificador, denominado *variable*, y cuyo concepto es uno de los pilares en la programación en los lenguajes imperativos. El lenguaje FORTRAN con su estructura matemática original es el precursor, véase figura 2.

¤ *Estructura de programas basada en instrucciones*. Un programa en un lenguaje imperativo se caracteriza por realizar sus tareas ejecutando iterativamente una secuencia de pasos elementales. Esto es una consecuencia de la arquitectura de *Von Neumann*, en la cual las instrucciones se almacenan en memoria, repitiendo determinadas secuencias de instrucciones. Así los mecanismos de control son abstracciones

del uso del contador de programa para localizar la instrucción siguiente y para realizar saltos (mediante la actualización del contador de programa). Los lenguajes COBOL y FORTRAN son los precursores, véase la figura 2.

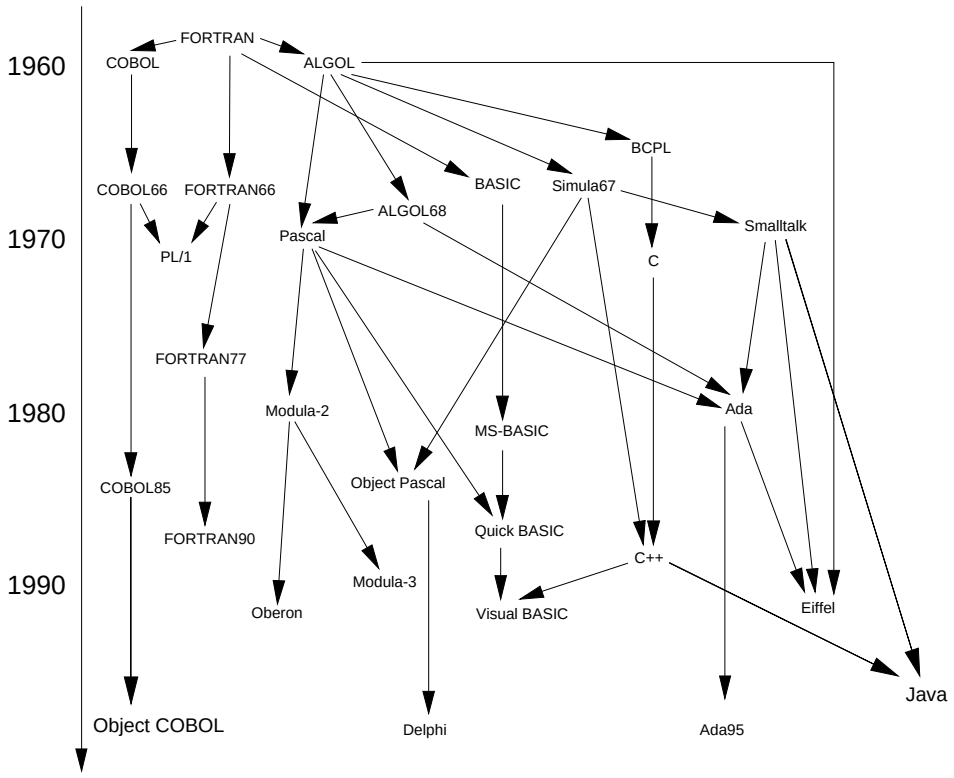


Fig. 2: La evolución de los lenguajes imperativos

- *Manejo frecuente de las instrucciones de asignación*. Ligado al concepto de variable y de memoria, hay la idea de que cada expresión calculada debe ser almacenada en memoria, para poder utilizarse posteriormente en otras instrucciones del programa. La instrucción de asignación ya está incorporada en la primera versión de FORTRAN.
- *Resolución de algoritmos por medio de estructuras de control secuenciales, alternativas y repetitivas*. En un principio la programación en lenguajes imperativos hacia uso intensivo de la instrucción *GOTO*, como consecuencia directa de la estructura de control interna de la arquitectura de *Von Neumann*. El artículo de *E.W. Dijstra* titulado "*Goto statement considered harmful*" [DIJS68] comenzó lo que más tarde se denominaría *Programación estructurada*, y provocó que los lenguajes incluyeran estructuras de control de flujo más avanzadas. La influencia del Pascal como vehículo de la enseñanza de la programación ha sido decisiva en este aspecto, tal y como se observa en la figura 2.
- *Incorporación de mecanismos de manejo de bloques*. La complejidad del software y las técnicas de diseño modular aconsejaron incorporar a los lenguajes imperativos mecanismos para el manejo de bloques. Estos bloques pueden tener variables locales, cuya existencia y visibilidad está restringida al bloque donde están declaradas. Los bloques pueden comunicarse entre ellos por medio de intercambio de información por valor o referencia. También se les ha dotado de recursividad. El precursor fue el lenguaje ALGOL y ALGOL68, propagándose al resto de los lenguajes como Pascal, C, Ada, etc... tal como se ve en la figura 2. Los lenguajes Modula-2 y Ada son los que han remarcado más este aspecto, y su influencia se ha propagado a diversos mecanismos de lenguajes posteriores o reformas de lenguajes existentes. Por ejemplo las *units* de Object Pascal y Turbo Pascal son repercusiones directas de los módulos de Modula-2 y los paquetes de Ada.

- *Gestión de memoria dinámica heap en tiempo de ejecución*. Los lenguajes imperativos se dotaron de mecanismos que permiten reservar y liberar memoria dinámica *heap* (montón). El precursor es el lenguaje ALGOL (véase figura 2), y el lenguaje que realiza más uso intensivo de este mecanismo es el lenguaje C, y su influencia se ha propagado a todos las revisiones de los lenguajes clásicos (FORTRAN90, Visual BASIC, etc...).
- *Adecuación al paradigma de orientación a objetos*. La mayor parte de los lenguajes imperativos están adaptándose al paradigma de orientación de objetos incorporando nuevas instrucciones para dar soporte a la encapsulación, herencia y polimorfismo. El precursor es el lenguaje Simula67, y su influencia está llegando a todos los lenguajes imperativos, tal y como se muestra en la figura 2. Incluso lenguajes tradicionales como COBOL y FORTRAN están preparando una revisión para introducir el mecanismo de orientación a objetos [GREH94].

§ ***Los lenguajes declarativos*** son lenguajes de muy alto nivel cuya notación es muy próxima al problema real del algoritmo que resuelven. Hay dos tipos de lenguajes declarativos: *lenguajes funcionales* y *lenguajes lógicos*.

- Los *lenguajes funcionales* o *aplicativos* tienen todas sus construcciones como llamadas a funciones matemáticas. Es decir no hay instrucciones, todo el programa es una función y todas las operaciones se realizan por composición de funciones más simples. En la ejecución de los programas funcionales se "aplica" la función a los datos de entrada (que son los argumentos de la función) y se obtiene el resultado (el valor calculado por la función). El precursor de los lenguajes funcionales ha sido la teoría *lambda calculus* (introducida por *Alonzo Church* en 1932) y el primer lenguaje de amplia difusión es el LISP creado por *John McCarthy* en 1959. Ejemplos de lenguajes funcionales: LISP [STEE90], CLOS [PAEP93], Scheme [IEEE91], APL, ML, Miranda [BIRD88], Hope [BAIL90], Haskell, Concurrent Clean, Erlang, etc...

  Hasta ahora los lenguajes funcionales se han utilizado poco en la industria debido a su bajo rendimiento y voracidad de memoria frente a los lenguajes imperativos. Sin embargo desde mediados de los años ochenta se está consiguiendo que los lenguajes funcionales se acerquen a los imperativos en tiempo de ejecución y en gestión de memoria eficiente [POUN94].
- Los *lenguajes lógicos* definen sus instrucciones siguiendo un tipo de Lógica. El lenguaje de programación lógica más utilizado es el PROLOG, que utiliza la lógica clausal (restringida a las cláusulas de Horn). La programación lógica maneja *relaciones* (predicados) entre objetos (datos). Las relaciones se especifican con *reglas* y *hechos*. La ejecución de programas lógicos consiste en la demostración de hechos sobre las relaciones por medio de preguntas. Ejemplos de lenguajes lógicos: PROLOG [STER94], Concurrent PROLOG, Prolog++, GHC, Parlog, Vulcan, Polka, etc...

§ ***Los lenguajes concurrentes*** son los que permiten la ejecución simultánea ("paralela" o "concurrente") de dos o varias tareas. La concurrencia puede ser una característica propia del lenguaje, o el resultado de ampliar las instrucciones de un lenguaje no concurrente. Ejemplos: Ada, Concurrent C, Concurrent Pascal, Concurrent Prolog, CSP, Argus, Actors, Linda, Monitors, Ccc32 compilador de C concurrente para Windows [GOME97a, GOME97b] etc...

§ ***Los lenguajes orientados a objetos (LOO)***. Actualmente existen más de 2.000 lenguajes de alto nivel diferentes, de los cuales alrededor de un centenar son lenguajes orientados a objetos o lenguajes basados en objetos. Un lenguaje de programación se dice que es un *lenguaje basado en objetos* si soporta directamente tipos abstractos de datos y clases (por ejemplo Ada80 y Visual Basic 4). Un *lenguaje* orientado a objetos es también basado en objetos, pero añade mecanismos para soportar la *herencia* (como un medio para soportar la jerarquía de clases) y el *polimorfismo* (como un mecanismo que permite relizar una acción diferente en función del objeto o mensaje recibido). Ejemplos: Smalltalk, C++, Object Pascal, Turbo Pascal, Delphi, CLOS, Prolog++, Java y Eiffel.

Bertrand Meyer, autor del lenguaje Eiffel, propone las siete características de un *lenguaje orientado a objetos puro* [MEYE88]:

1ª) Estructua modular basada en clases

2ª) Abstracción de datos. Los objetos se describen como implementaciones de tipos abstractos de datos.

3ª) Gestión de memoria automática. El lenguaje libera automáticamente de la memoria los objetos no utilizados. Es decir no existen instrucciones o funciones como dispose(), delete, free(), salvo para objetos especiales.

4ª) Clases. Todo tipo de datos es una clase, excepto los tipos simples. No existen funciones o procedimientos libres, todos son métodos de una clase.

5ª) Herencia. Una clase puede definirse como una extensión o una restricción de otra.

6ª) Polimorfismo y enlace dinámico. Las entidades pueden referenciar a objetos de más de una clase, y se pueden realizar diferentes operaciones en clases diferentes.

7ª) Herencia múltiple y repetida. Una clase puede heredar de más de una clase y ser padre de la misma clase más de una vez. Esta característica no es soportada por todos los lenguajes orientados a objetos puros. Así Smalltalk y Java sólo tienen herencia simple, aunque Java soporta la herencia múltiple de interfaces.

Ejemplos de LOO puros son Smalltalk, Eiffel y Java.

Por contraposición a los LOO puros están los denominados *lenguajes orientados a objetos híbridos* que son los lenguajes que tienen tipos orientados a objetos (clases) y tipos no orientados a objetos (tipos primitivos), algunos ejemplos de este tipo de lenguajes son Object Pascal, Turbo Pascal, Delphi y C++. La mayor parte de estos lenguajes han nacido como extensión de otros lenguajes no orientados a objetos. Un esquema sobre la evolución y semejanzas de los distintos LOO se presenta en la figura 3.

Otra clasificación de los LOO es *LOO basados en clases* y *LOO basados en prototipos* [EVIN94].

Los *LOO basados en clases* son los lenguajes cuyo sistema de tipos está basado en clases. Ejemplos de LOO basados en clases son C++, Objective-C, Smalltalk, Object Pascal, Java, CLOS y Eiffel.

Los *LOO basados en prototipos* no tienen clases definidas explícitamente, son sustituidas por objetos denominados prototipos. Para crear nuevas instancias se utiliza la clonación. Los principios básicos de un LOO basado en prototipos son los *prototipos* y la *delegación*. Un objeto prototipo se utiliza como un modelo para crear otro objeto. La delegación es un mecanismo por el cual el código a ejecutar se selecciona dinámicamente en tiempo de ejecución. Ejemplos de lenguajes basados en prototipos: Self [UNGA87, CHAM92, UNGA92, SELFi], T [SLAD87], ObjectLisp (es una extensión de Common Lisp, que a diferencia del CLOS -basado en clases- utiliza prototipos y paso de mensajes), Cecil (es un descendiente del Self, fue creado por *Craig Chambers*, uno de los diseñadores de Self), NewtonScript® (desarrollado por Apple® para su modelo Newton® [SWAI94a, SWAI94b, SWAI94c, SWAI94d]) y PROXY [LEAV93].

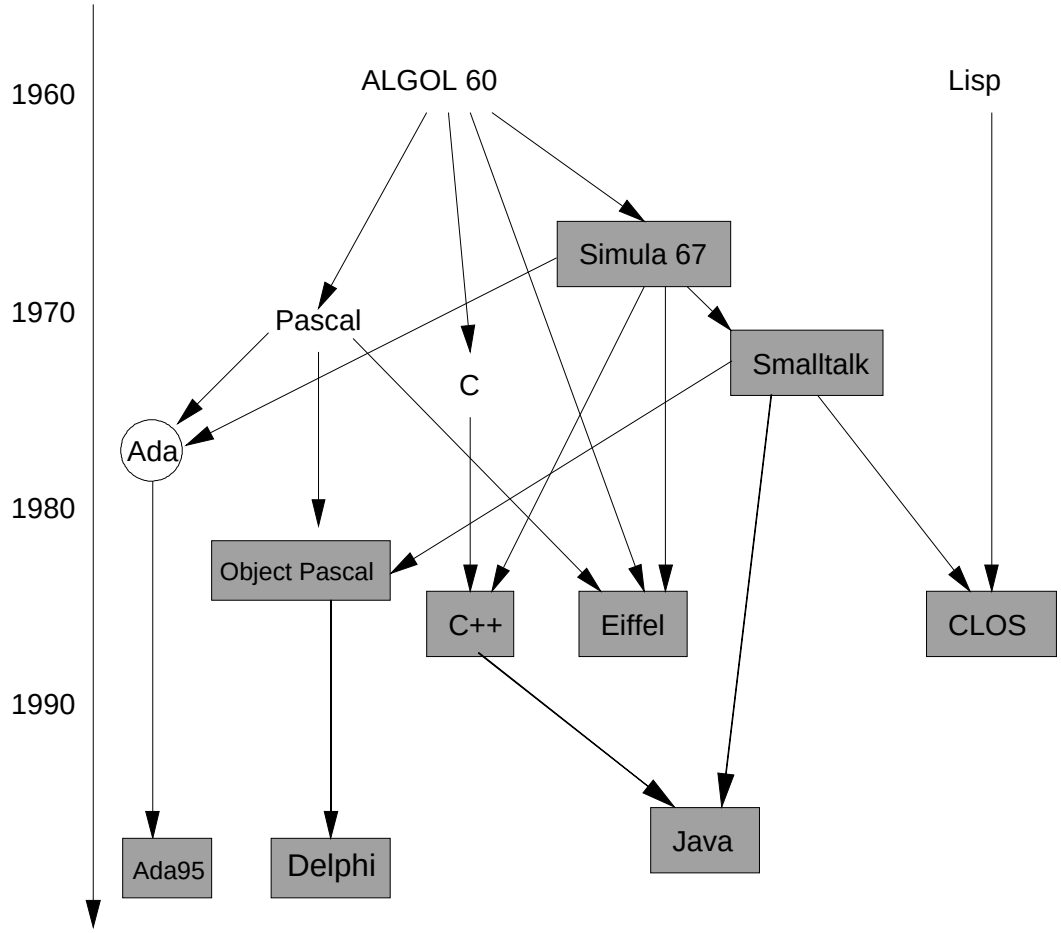


Fig. 3: Evolución de los lenguajes orientados a objetos

• Por generaciones

La Informática es una ciencia joven, nació en los años cuarenta, y tomó su carácter de disciplina académica en los años sesenta con la creación de los primeros centros de Cálculo en la Universidades. La evolución de los lenguajes de programación ha ido, en cierta medida, paralelo al desarrollo de la Informática. Los lenguajes se pueden clasificar según generaciones en:

- *Primera generación*
- *Segunda generación*
- *Tercera generación*
- *Cuarta generación*
- *Quinta generación*
- *Generación orientada a objetos*
- *Generación visual*
- *Generación internet*

  - La ***primera generación*** está constituida por los lenguajes máquina y ensamblador desarrollados en los años cuarenta y cincuenta.
  - La ***segunda generación*** comienza con el desarrollo del primer compilador de FORTRAN en el año 1958 y continua hasta mediados de los años sesenta con la normalización por el *American National Standard Institute (ANSI)* de los lenguajes de programación, ligados hasta ese momento a marcas comerciales. Se caracteriza por los lenguajes con asignación estática de memoria, es decir toda la memoria se asigna en tiempo de compilación. Estos lenguajes no tenían ni recursividad, ni manejaban estructuras dinámicas de datos. Ejemplos de estos lenguajes son FORTRAN y COBOL.
  - La ***tercera generación*** está ligada al término *programación estructurada*, y se desarrolla entre mediados de los años sesenta y mediados de los años setenta, aunque a nivel de lenguajes sus raices están en el lenguaje Algol 60 (desarrollado a principios de los años sesenta). Como características principales de los lenguajes de esta generación son: uso de módulos o subprogramas, variables locales, recursividad, y estructuras dinámicas. Ejemplos de estos lenguajes son: Algol 60, PL/I, Algol 68, Pascal, Modula, y C.
  - La ***cuarta generación*** está caracterizada por lenguajes de muy alto nivel dedicados a tareas específicas, en muchos casos denominados herramientas. Una gran parte de ellos están dedicados a la gestión de bases de datos y a la generación de aplicaciones con herramientas CASE. El desarrollo de estos lenguajes se produce desde mediados de los años setenta hasta finales de los ochenta. Ejemplos: SQL, DB2, DBASE, Ingress, Natural, Ideal, Application Factory, etc...
  - La ***quinta generación*** está asociada a los lenguajes ligados a la Inteligencia Artificial. La mayor parte de este tipo de lenguajes son versiones actualizadas o descendientes de los lenguajes Lisp y Prolog. Aunque el lenguaje Lisp data de 1959, el desarrollo intenso de estos lenguajes se produce desde principios de los años ochenta a principios de los noventa. Ejemplos: Common Lisp, Prolog, Parlog, ML, Haskell, Miranda, etc...
  - ***Generación orientada a objetos***. Es la última generación puede decirse que comienza a mediados de los años ochenta, aunque el primer lenguaje orientado a objetos fue el Simula 67, presentado en el año 1967 (fig. 2). Smalltalk se comenzó a desarrollar a principios de los años setenta, desembocando en el Smalltalk 80, de amplio uso en la decada de los años ochenta. El desarrollo del lenguaje Ada (basado en objetos, no orientado a objetos) a finales de los 70 y la proliferación de las Interfaces Gráficas de Usuario (IGU) a finales de los ochenta ha implicado la creación de nuevos lenguajes o la ampliación de lenguajes para introducir el paradigma de orientación a objetos. Ejemplos: Simula, Smalltalk, C++, Object Pascal, Turbo Pascal, CLOS, ObjectLisp, Eiffel, y Oberon.
  - ***Generación visual***. Comienza a principios de los años noventa, ligado a la exigencia de los usuarios de disponer de interfaces amigables. La mayor parte de las características visuales van orientadas al diseño del interfaz de usuario. Ejemplos: Visual Basic®, Delphi® (versión visual del lenguaje Object Pascal),...
  - ***Generación internet***. Comienza a mediados de los años noventa, ligado a la gran explosión de internet y a la necesidad de manejar aplicaciones en distintas plataformas dentro de toda la red. El lenguaje más característico es Java, pero también se podrían incluir dentro de esta generación: XML, HTML, VRML y otros.

1.2 Ventajas de los lenguajes de alto nivel

Las ventajas de los lenguajes de alto nivel sobre los de bajo nivel (lenguaje máquina y lenguaje ensamblador) son las siguientes:

- Los lenguajes de alto nivel son *más fáciles de aprender* que los de bajo nivel. El aprendizaje de muchos lenguajes de alto nivel necesita pocos o nulos conocimientos del hardware, ya que dichos lenguajes son prácticamente independientes de la máquina. Sin embargo los lenguajes de alto nivel están cerrados a ciertas zonas de la máquina, a las que tienen acceso los lenguajes de bajo nivel.
- Los programadores *se liberan de ocupaciones rutinarias* con referencias a instrucciones simbólicas o numéricas, asignaciones de memoria, etc...Tales tareas se manejan por un traductor que traslada el lenguaje de alto nivel al lenguaje máquina, y se evitan errores humanos.

- Un programador de lenguajes de alto nivel *no necesita conocer la forma en que se colocan los distintos tipos de datos en la memoria del ordenador*.
- La mayoria de los lenguajes de alto nivel ofrecen al programador una *gran variedad de estructuras de control*, que no se disponen en los lenguajes de bajo nivel. Por ejemplo los lenguajes de alto nivel tienen las siguientes construcciones de control de flujo:
    - Sentencias condicionales (tales como IF-THEN-ELSE o CASE)
    - Bucles (FOR, WHILE, REPEAT)
    - Estructuras de bloque (BEGIN-END)

  Estas estructuras de programación fomentan estilos y técnicas de programación estructurada. Obteniéndose programas fáciles de leer, comprender y modificar. Con lo que se reducen los costos de programación al ser programas menos complicados que los realizados en lenguajes de bajo nivel.
- Los programas escritos en lenguajes de alto nivel *se depuran más fácilmente* que los escritos en lenguaje máquina o ensamblador. Los lenguajes de alto nivel tienen construcciones que eliminan o reducen ciertos tipos de errores de programación, que aparecen en los lenguajes de bajo nivel.
- Los lenguajes de alto nivel tienen una *mayor capacidad de creación de estructuras de datos*, tanto estáticas como dinámicas, que facilitan la resolución de muchos problemas. Los lenguajes orientados a objeto también pueden manejar tipos abstractos de datos, que permiten la reutilización de código.
- Los lenguajes de alto nivel permiten un *diseño modular de los programas*, con lo que se consigue una división del trabajo entre los programadores, obteniéndose un mejor rendimiento en la realización de aplicaciones.
- Los lenguajes de alto nivel también incluyen la posibilidad de incorporar *soporte para la programación orientada a objetos y la genericidad*.
- Finalmente los lenguajes de alto nivel son relativamente *independientes de la máquina*, obteniéndose programas que se pueden ejecutar en diferentes ordenadores. Tan sólo hay que volver a compilarlos o interpretarlos, reduciéndose costos y, en general, pueden instalarse de nuevo si se queda el ordenador obsoleto. Sin embargo todos los programas escritos en lenguaje máquina y en ensamblador no se pueden implementar en otros ordenadores que no posean una arquitectura compatible o se utice algunos de los métodos de transporte que se verán en el apartado 3.

Un lenguaje de alto nivel ha de procesarse antes de ejecutarse para que el ordenador pueda comprenderlo. Este proceso se realizará mediante un *procesador de lenguaje*.

Dada la importancia de poseer lenguajes de distintos niveles, se entiende la necesidad de escribir traductores que trasladen los lenguajes entre sí; de compiladores que pasen los lenguajes de alto nivel a bajo nivel; y de ensambladores que generen el código máquina de las distintas arquitecturas de ordenadores. A los traductores, compiladores e intérpretes, ensambladores, y otros programas que realizan operaciones con lenguajes se les denomina con el nombre genérico de **procesadores de lenguaje**.

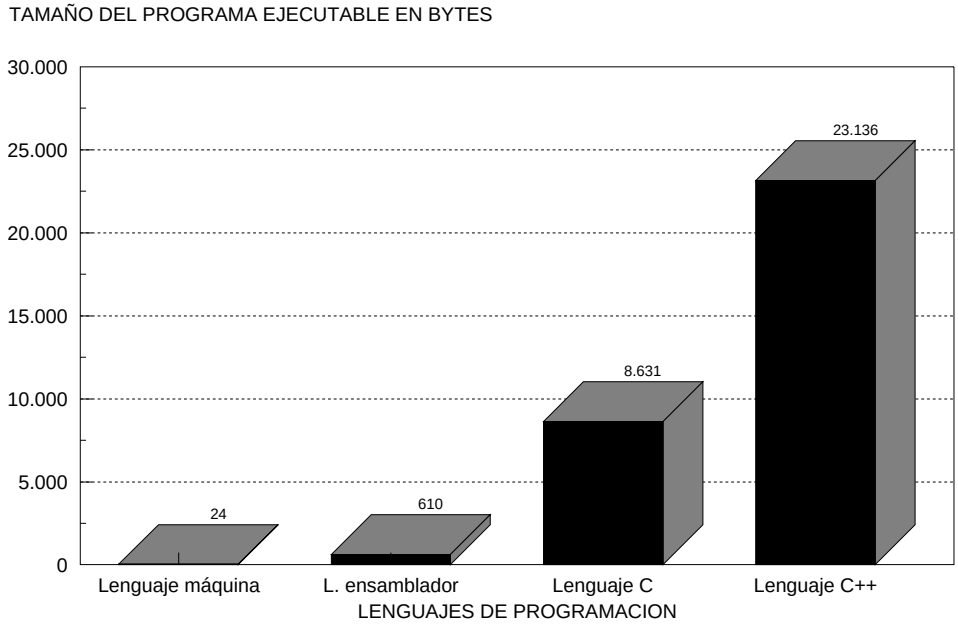


Fig. 4: Niveles de los lenguajes de programación y tamaño de los ficheros ejecutables

1.3 Inconvenientes de los lenguajes de alto nivel

Los lenguajes de alto nivel también tienen sus inconvenientes sobre los lenguajes de bajo nivel. La comodidad del programador se *paga* en mayores tamaños de los ficheros ejecutables, y consecuentemente en mayores tiempos de compilación y de ejecución. Así en los anexos I (lenguaje máquina 80x86), II (lenguaje ensamblador 80x86), IV (lenguaje C) y VIII (lenguaje C++) se han escrito sendos programas que presentan en pantalla una cadena de caracteres con un saludo. Los cuatro ficheros ejecutables realizan la misma operación, mostrar en pantalla el mensaje "*Hola a todos*", sin embargo el tamaño del fichero ejecutable se va incrementando según se va ascendiendo en el nivel del lenguaje (fig. 4). El lector puede comprobar que los tiempos de compilación y ejecución también se van incrementando según se va ascendiendo en el nivel del lenguaje. Sin embargo las ventajas enumeradas en el apartado anterior suelen superar con creces, en la mayor parte de las aplicaciones, los inconvenientes enunciados anteriormente.

1.4 Otros lenguajes

Los lenguajes manejados en Informática pueden tener otras aplicaciones distintas de la programación [VALD91], a continuación se presenta algunos de ellos:

- *Lenguajes de descripción de página*: Postscript®, PCL®, ESC/2®, True Page®,...
- *Formatos gráficos raster*: TIFF, GIF, PCX, BMP, Photo CD, JPEG, TGA,...
- *Formatos gráficos vectoriales*: CGM, DXF, WGM,...
- *Formatos de bases de datos*: DBF, DBT, NDX, MDX,...
- *Formatos de texto*: RTF, ASCII, Word®, WordPerfect®,...
- *Formatos de archivos de sonido*: WAV, MIDI, etc...
- *Formatos de archivos de video*: AVI, MOV, etc...
- *Formatos de ayuda*: HLP de Windows®, HLP de Turbo Vision®,...
- *Lenguajes de gestión electrónica de documentos o hipertexto*: formato pdf de Acrobat®, Hypertext Markup Language (HTML) [MCAR94], XML,...
- *Lenguajes multimedia o de autor*: Multimedia Toolbook®, MediaDeveloper®, Macromind Director®, HyperStudio®, Multimedia Workshop®, Multimedia MasterClass®, Q/Media®, etc...
- *Lenguajes de generación de imágenes*: POV, etc...
- *Lenguajes de animación de imágenes*: ANIMA Object [JUAN94, CUEV94b], MiraCine [THAL90], etc...
- *Lenguajes naturales*: subconjuntos del Castellano, Inglés, etc...

2. PROCESADORES DE LENGUAJE

Procesadores de lenguaje *es el nombre genérico que reciben todas las aplicaciones informáticas en las cuales uno de los datos fundamentales de entrada es un lenguaje*. La definición anterior afecta a una gran variedad de herramientas software, algunas de ellas son las siguientes (expresando entre paréntesis el término equivalente en lengua inglesa):

- Traductores (*translators*)
- Compiladores (*compilers*)
- Ensambladores (*assemblers)*
- Montadores de enlaces o enlazadores (*linkers*)
- Cargadores (*loaders*)
- Intérpretes (*interpreters*)
- Desensambladores (*dissemblers)*
- Decompiladores (*decompilers*)
- Depuradores (*debuggers*)
- Analizadores de rendimiento (*profilers*)
- Optimizadores de código (*code optimizers*)
- Compresores (*compressors*)
- Preprocesadores (*preprocessors*)
- Formateadores (*formatters*)
- Editores (*editors*)

2.1 Traductores

Un **traductor** *es un programa que procesa un texto fuente y genera un texto objeto*. El traductor está escrito en un *lenguaje de implementación (LI)* o también denominado lenguaje *host*. El texto fuente está escrito en *lenguaje fuente (LF)*, por ejemplo un lenguaje de alto nivel. El texto objeto está escrito en *lenguaje objeto (LO)*, por ejemplo un lenguaje máquina, ensamblador u otro lenguaje de alto nivel. Se suele utilizar la notación en T, tal como se muestra en la figura 5. La notación en T también se puede representar de una forma abreviada como: $LF_{LI}LO$.

El *lenguaje fuente (LF)* es el lenguaje origen que transforma el traductor (por ejemplo C, C++, Pascal, FORTRAN, PL/I, COBOL, ADA, MODULA-2, BASIC, C..). También pueden ser lenguajes de bajo nivel.

El *lenguaje objeto (LO)* es el lenguaje al que se traduce el texto fuente. Los lenguajes objeto pueden ser por ejemplo otro lenguaje de alto nivel, el lenguaje máquina de un microprocesador determinado, un lenguaje ensamblador,...

El *lenguaje de implementación (LI) o lenguaje host* es el lenguaje en que está escrito el traductor. Puede ser cualquier lenguaje, desde un lenguaje de alto nivel a un lenguaje máquina.

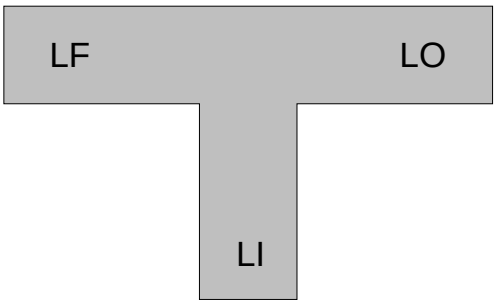


Fig. 5: Notación en T

2.2 Ensambladores

Si el lenguaje fuente es el lenguaje ensamblador (en inglés *assembly*) *y el lenguaje objeto es el lenguaje máquina, entonces al traductor se le llama* **ensamblador** (en inglés *assembler*). Los ensambladores son traductores sencillos, en los que el lenguaje fuente tiene una estructura simple, que permite una traducción de una sentencia fuente a una instrucción en lenguje máquina, guardándose en casi todos los casos esta relación uno a uno. Hay ensambladores que tienen macroinstrucciones en su lenguaje. Estas macroinstrucciones, de acuerdo con su nombre, se suelen traducir a varias instrucciones de máquina. A este tipo de ensambladores se les denomina **macroensambladores** (en inglés *macroassembler*). Actualmente la mayor parte de los ensambladores comerciales son macroensambladores.

2.3 Compiladores

Un traductor que transforma textos fuente de lenguajes de alto nivel a lenguajes de bajo nivel se le denomina **compilador** (en inglés *compiler*).

El tiempo que se necesita para traducir un lenguaje de alto nivel a lenguaje objeto se denomina **tiempo de compilación** (*compilation time*). El tiempo que tarda en ejecutarse un programa objeto se denomina **tiempo de ejecución** (*run time*).

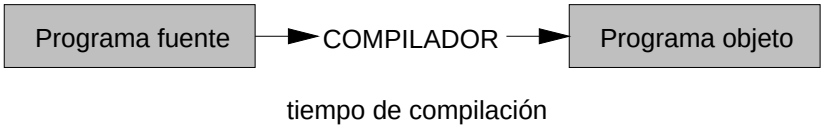


Fig. 6: Concepto de tiempo de compilación

Nótese que el programa fuente y los datos se procesan en diferentes momentos, denominados tiempo de compilación y tiempo de ejecución.

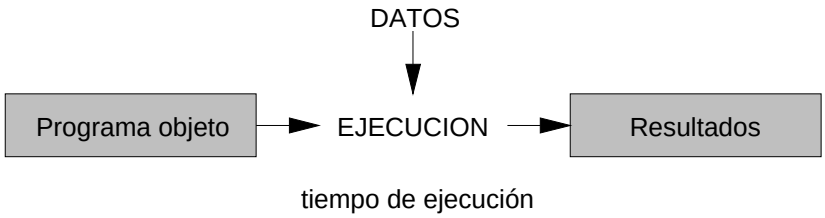


Fig. 7: Concepto de tiempo de ejecución

2.4 Montadores de enlaces

Entre el proceso de compilación y la ejecución existe el proceso de montaje de enlaces, que se produce cuando el lenguaje fuente permite una fragmentación de los programas en trozos, denominados de distintas formas según el lenguaje de programación empleado (módulos, units, librerias, procedimientos, funciones, subrutinas..., el lenguaje ADA introduce la unidad de compilación, en inglés *compilation unit*). Dichas partes o trozos pueden compilarse por separado, produciendose los códigos objetos de cada una de las partes. El montador de enlaces o enlazador (en inglés *linker*) realiza el montaje de los distintos códigos objeto, produciendo el módulo de carga, que es el programa objeto completo, siendo el cargador (en inglés *loader*) quien lo trasfiere a memoria (fig. 8).

La compilación genera un código objeto llamado **reubicable**, cuyo significado es que las posiciones de memoria que utiliza son relativas. El montador de enlaces se encarga de colocar detrás del programa principal del usuario, todos los módulos reubicables.

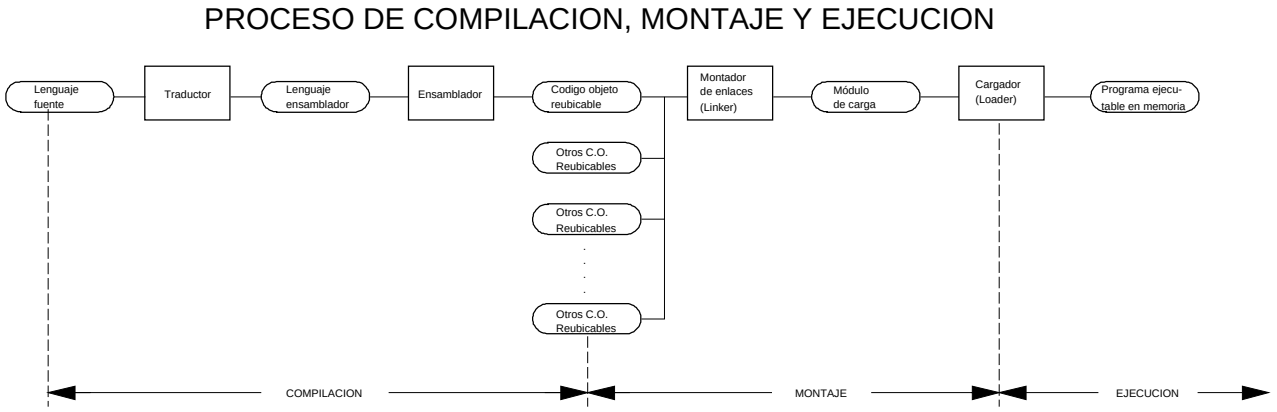


Fig. 8: Fases de compilación, montaje y ejecución

2.5 Cargadores

El cargador se encarga de colocar el fichero ejecutable en memoria, asignando el espacio necesario al programa en memoria, y pasando el control a la primera de las instrucciones a ejecutar, comenzando a continuación la fase de ejecución. El cargador es un programa incluido con el sistema operativo. Así en el sistema operativo MS-DOS® el cargador está dentro del intérprete de comandos COMMAND.COM. Véase en el anexo XV una explicación de como trabaja el cargador en el sistema operativo MS-DOS®.

2.6 Intérpretes

Los intérpretes son programas que simplemente ejecutan las instrucciones que encuentran en el texto fuente. En muchos casos coexisten en memoria el programa fuente y el programa intérprete (fig. 9). Nótese que en este caso todo se hace en tiempo de ejecución. Algunos de los lenguajes comúnmente interpretados son el BASIC, LOGO, PROLOG, SMALLTALK, APL y LISP.

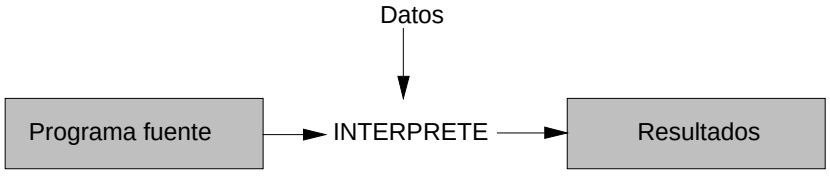


Fig. 9: Funcionamiento de un intérprete

Evidentemente la ejecución de un programa compilado será más rápida que la del mismo programa interpretado. Sin embargo los intérpretes son más interactivos y facilitan la puesta a punto de programas. Algunos lenguajes de programación tan sólo pueden ser interpretados debido a sus características, por ejemplo algunos lenguajes funcionales y lenguajes orientados a objeto. En estos casos existen **intérpretes con compiladores incrementales** que pueden recompilar los módulos modificados en tiempo de ejecución.

Los intérpretes se estudian en el apartado 13.

2.7 Decompiladores

Los decompiladores realizan la tarea inversa a los compiladores, es decir son un caso particular de los traductores en los cuales el programa fuente es un lenguaje de bajo nivel y el lenguaje objeto es un lenguaje de nivel superior. Esta tarea es difícil, sobre todo si se desea que el lenguaje objeto tenga una cierta lógica, a no ser que se conozca la forma en que se obtuvo el programa de bajo nivel (por ejemplo si se generó con un compilador determinado, y se tiene el esquema de generación de código de dicho compilador).

2.8 Desensambladores

Un caso particular de los decompiladores son los desensambladores, que *traducen de código máquina a ensamblador*, quizá es un caso más fácil dado que hay una correspondencia directa entre las instrucciones ensamblador y código máquina.

Un ejemplo de desensambladores es el *j*-- [AZAÑ97, AZAÑ98], donde se desarrollan un conjunto de herramientas para desensamblar y ensamblar *bytecode* de la máquina virtual de Java (JVM).

2.9 Depuradores

Los depuradores (en inglés *debuggers*, literalmente desparasitadores) *son herramientas que permiten encontrar y corregir los errores de los programas* (denominádos en inglés *bugs*, parásitos). Estas herramientas suelen ir ligadas a los compiladores de forma que el programador pueda comprobar y visualizar la correcta ejecución de un programa. Las características habituales de los depuradores son:

- Permiten observar la traza de los programas fuente, permitiendo la visualización del valor de cualquier variable, dirección, o expresión.
- Comprobación del código objeto generado (habitualmente ensamblador o instrucciones en código máquina) por cada instrucción del programa fuente.
- Observación de la traza a bajo nivel del programa ejecutable, visualizando en ejecución los valores de las distintas posiciones de memoria, de los registros del microprocesador, etc...

Los depuradores utilizan parte de la información usada en tiempo de compilación por el compilador, y que habitualmente no se almacena en ejecución, lo cual permite restablecer los lazos existentes entre el código fuente y el código objeto. Como depuradores comerciales en el entorno DOS se pueden citar *TURBO DEBUGGER*® de Borland y *CodeView*® de Microsoft.

2.10 Analizadores de rendimiento

Los analizadores de rendimiento (denominados habitualmente en lengua inglesa *profilers*) son herramientas que permiten examinar el comportamiento de los programas en tiempo de ejecución, permitiendo comprobar que zonas del código trabajan eficientemente y cuales deberían ser revisadas por su bajo rendimiento. Actualmente la mayor parte de los compiladores comerciales incorporan analizadores de rendimiento, por ejemplo: *TURBO PROFILER*® de Borland y *Source Profiler*® de Microsoft.

2.11 Optimizadores de código

Los optimizadores de código pueden ser herramientas independientes, o estar incluidas en los compiladores e invocarse por medio de opciones de compilación, siendo esta última forma como se suelen encontrar en la mayor parte de los compiladores comerciales.

Una opción habitual de optimización es elegir entre velocidad de ejecución y tamaño del código ejecutable. Otras opciones son: generar código para un microprocesador específico dentro de una familia de microprocesadores, eliminar la comprobación de rangos o desbordamientos de pila (*stack*), evaluación en cortocircuito (*short-circuit*) para expresiones booleanas, eliminación de código muerto o no utilizado, eliminación de funciones no utilizadas (por ejemplo la opción /PACKFUNCTIONS de Microsoft®), etc...

2.12 Compresores

Los compresores de ficheros son una herramienta habitual (PKZIP, ARJ, ...) de uso en el campo de la informática. Un caso particular son los compresores de ficheros ejecutables que reducen el tamaño de los ejecutables, existen varios en el mercado: por ejemplo la opción /EXEPACK de Microsoft® para los programas desarrollados con compiladores de Microsoft, y otros de uso con cualquier ejecutable (PKLITE, LZEXE, etc...).

2.13 Preprocesadores

Es un caso particular de un traductor en el cual se hacen sustituciones de las macros definidas. El preprocesador realiza las sustituciones, pero no hace ningún tipo de análisis del contexto donde las realiza, ésta es la principal diferencia entre un preprocesador y otros tipos de procesadores de lenguaje. Un ejemplo de preprocesador es el incorporado por los compiladores de lenguaje C.

2.14 Formateadores

Los formateadores pueden ser de muchos tipos y con diferentes fines, desde los dedicados a formatear textos, ecuaciones o programas. Los formateadores de programas resaltan su sintaxis o su estructura, para lo cual es necesario conocer la sintaxis del lenguaje a formatear. También pueden entrar dentro de este grupo los conversores de formatos.

2.15 Editores

Los editores de lenguajes de programación con sintaxis resaltada por medio de colores o tipos de letra permiten llamar a la atención al programador en el momento mismo que está escribiendo el programa, sin necesidad de compilar, dado que llevan incorporada la sintaxis del lenguaje.

3. ARQUITECTURAS DE ORDENADORES

En la disciplina de Procesadores de Lenguaje los compiladores ocupan una posición privilegiada, dado que son las herramientas más utilizadas por los profesionales de la Informática para el desarrollo de aplicaciones. En el caso particular del desarrollo de compiladores es necesario tener definidos tres pilares básicos:

- *La definición léxica, sintáctica y semántica del lenguaje fuente a compilar*
- *La estructura interna del compilador*
- *La arquitectura del ordenador y su repertorio de instrucciones que constituirá el lenguaje objeto*

Los dos primeros pilares se tratarán en los apartados siguientes, con el tercer pilar tan sólo trata de resaltarse la relación intima entre las tecnologias de compilación y las arquitecturas de ordenadores, y los problemas que plantea la rápida evolución de éstas últimas [LILJ94].

El desarrollo de nuevas arquitecturas de ordenadores que origina a su vez las correspondientes generaciones de microprocesadores con sus repertorios de instrucciones planteándose entre otros estos dos problemas:

- *¿Cómo se pueden ejecutar las aplicaciones desarrolladas para otras arquitecturas de ordenadores en la nueva arquitectura?*
- Dado que un compilador es un programa demasiado complejo para escribirlo directamente en lenguaje máquina *¿Con qué se escribe el primer compilador?*

Las dos cuestiones anteriores están ligadas entre sí, sin embargo el primer problema planteado no sólo es aplicable a la construcción de nuevas arquitecturas, también se aparece cuando es necesaria la compatibilidad de aplicaciones entre diferentes sistemas operativos y arquitecturas de ordenadores. Las soluciones habitualmente empleadas para resolver este problema es el empleo de alguna de las técnicas y herramientas presentadas en la fig. 10 y que se comentan a continuación [SITE93].

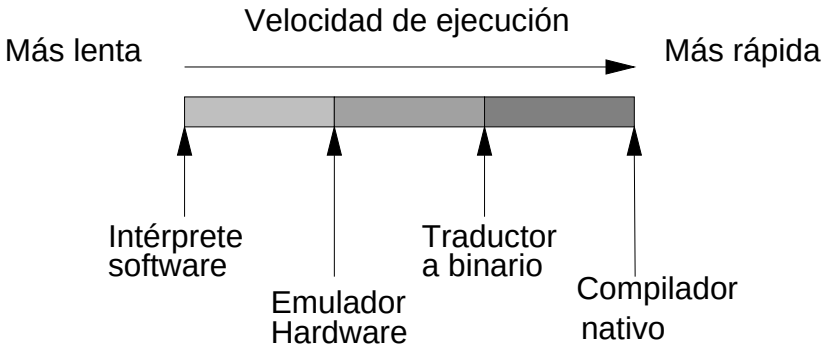


Fig. 10: Técnicas para realizar migraciones de aplicaciones entre distintas arquitecturas

- *Intérpretes software o emuladores software*. Un intérprete software de código binario es un programa que lee una a una las instrucciones binarias de la arquitectura antigua que están en un fichero ejecutable, y las interpreta. Los intérpretes no son muy rápidos, pero se pueden construir y adaptar a distintas arquitecturas sin excesivos costes de desarrollo. Se puede aumentar el rendimiento de los intérpretes creando una caché que permita almacenar formas intermedias de instrucciones que ya se interpretaron previamente. Ejemplos comerciales de intérpretes software binario son los emuladores de DOS para distintos sistemas operativos (por ejemplo SoftPC® de Insignia Solutions). Otro ejemplo son los intérpretes de la máquina abstracta JVM (Java Virtual Machine) para distintas plataformas y que permiten ejecutar los códigos binarios denominados *bytecode* (ficheros con la extensión *.class*).

- *Emuladores hardware*. Un emulador hardware trabaja de forma similar a un intérprete de software, pero está implementado en hardware de forma que decodifica las instrucciones de la arquitectura antigua y las traduce a la nueva arquitectura. Un emulador hardware es mucho más rápido que un intérprete software, sin embargo sólo se puede diseñar para una máquina específica. Un ejemplo son los microprocesadores Java que emulan la máquina abstracta JVM (Java Virtual Machine), también denominados por Sun arquitecturas PicoJava [WAYN96].
- *Traductores entre códigos binarios o ensambladores*. Son conjuntos de instrucciones de la nueva arquitectura que reproducen el comportamiento de un programa en la arquitectura antigua. Habitualmente, la información de la máquina antigua se almacena en registros de la nueva máquina. Los programas traducidos a nivel binario o ensamblador son más rápidos que los intérpretes software o emuladores hardware, pero más lentos que los programas construidos con compiladores nativos, que aprovechan al máximo la arquitectura de la nueva máquina. Ejemplos de tradcutores binarios son los desarrollados por DEC para traducir instrucciones de las arquitecturas VAX® y MIPS® a la nueva arquitectura ALPHA® [SITE93].
- *Compiladores nativos*. Los programas fuente antiguos se pueden volver a recompilar com compiladores desarrollados para la nueva arquitectura, aprovechando al máximo sus características. Esta es la opción que produce la mejor calidad de código objeto en la nueva arquitectura, y por consiguiente la mayor velocidad con respecto a las opciones comentadas anteriormente. En el siguiente apartado se explican los *compiladores cruzados*, como una herramienta para portar compiladores y software entre distintas arquitecturas.

Otro problema clásico en arquitectura de ordenadores es como medir el rendimiento de las diferentes arquitecturas y compararlas entre sí [PATT90]. Actualmente las pruebas de rendimiento más ampliamente aceptadas son las pruebas SPEC, desarrolladas por el organismo *System Performance Evaluation Corporation* y que están especializadas en medir el rendimiento en plataformas que utilizan microprocesadores diferentes. Una versión de las pruebas data de 1992, y con versiones diferentes para operaciones con enteros (SPECint92) y para operaciones con punto flotante (SPECfp92) [SHAR94].

4. COMPILADORES CRUZADOS

Uno de los problemas que se plantea es desarrollar los primeros compiladores en una arquitectura nueva, dado que un compilador es un programa demasiado complejo para escribirlo directamente en lenguaje máquina.

Sin embargo cuando se trabaja con un nuevo tipo de microprocesador, alguien tiene que escribir el primer compilador. Otro problema que se plantea es como se puede ejecutar el código antiguo de otras arquitecturas en la nueva arquitectura.

Sin embargo esta tarea se puede realizar más cómodamente en una máquina donde ya se dispongan herramientas software, los compiladores cruzados se encargaran del resto del trabajo.

Se denomina **compilador cruzado** (en inglés *cross-compiler*) *a un compilador que se ejecuta en una máquina pero el código objeto es para otra máquina*.

Supóngase que se escribe un compilador cruzado para un nuevo lenguaje L, siendo el lenguaje de implementación S, y el lenguaje objeto es el código de una máquina N (*nueva*). Usando una notación en T, se tiene:

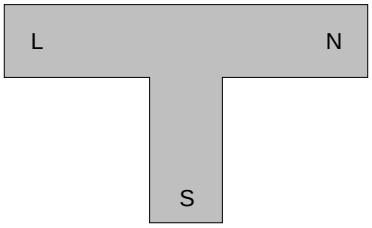


Fig. 11: Notación en T del compilador cruzado L_SN

Si ya existe un compilador del lenguaje S, que se ejecuta en una máquina M y genera un código para la máquina M, es decir usando la notación en T :

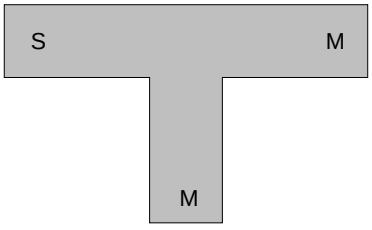


Fig. 12: Notación en T del compilador S_MM

Si L_SN se ejecuta a través de S_MM, entonces se tiene un compilador L_MN , es decir un compilador de L a N que se ejecuta en M. Este proceso se puede ilustrar en la figura 13, que muestra juntos los diagramas en T de estos compiladores.

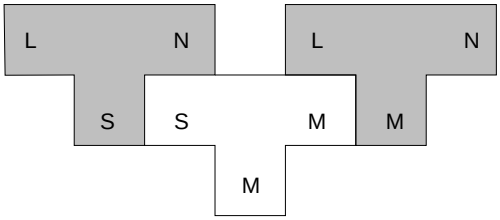


Fig. 13: Compilando un compilador cruzado

Cuando los diagramas en T se colocan juntos como en la figura 13, nótese que el lenguaje de implementación S del compilador L_SN debe ser el mismo que el lenguaje fuente del compilador ya existente S_MM, y además el lenguaje objeto M del compilador ya existente debe ser el mismo que el lenguaje de implementación de L_MN. Los tres diagramas en T de la figura 13 pueden ponerse en forma de ecuación:

$$L_SN + S_MM = L_MN$$

Ejemplo 4.1: Traductor cruzado Z80-80x86

A. González González en su Proyecto Fin de Carrera de la E.U. de Informática de Oviedo [GONZ91], construye un traductor cruzado entre ensamblador para el microprocesador Z80 y los microprocesadores de la familia 80x86.

Ejemplo 4.2: Preprocesador EQN

Las herramientas que acompañan al sistema operativo UNIX fueron unas de las primeras que utilizaron la tecnología de los procesadores de lenguaje y de los compiladores cruzados para su migración entre las distintas plataformas en las que se instaló el sistema operativo UNIX. Así el EQN de *Kernighan* y *Cherry* [KERN75] es un preprocesador de expresiones matemáticas, que permite la escritura de expresiones con operadores como *sub* y *sup* para subíndices y superíndices y genera los comandos para el formateador de textos TROFF. Si el preprocesador EQN encuentra un texto de entrada de la forma:

```
CAJA sup 2
```

y escribirá:

$$CAJA^2$$

El operador *sub* es similar sólo que para subíndices.

Estos operadores se pueden aplicar recursivamente, así por ejemplo el siguiente texto:

```
a sup {(i+k) sup {(2+j sub m)}}
```

resulta:

$$a^{(i+k)^{(2+j_m)}}$$

Las entradas al preprocesador EQN se tratan como si fuesen un lenguaje de programación y se aplica la tecnología de los traductores, con lo que se consiguen los siguientes beneficios:

- Fácil implementación
- Posibilidad de evolución del lenguaje

La primera versión del compilador EQN se escribió en lenguaje C, y generaba los comandos para el formateador de texto TROFF . En el siguiente diagrama se muestra un compilador cruzado para EQN, ejecutado sobre un ordenador PDP-11 de DEC®, así se obtuvo la ejecución de EQN_CTROFF a través del compilador de C del PDP-11 ($C_{11}11$).

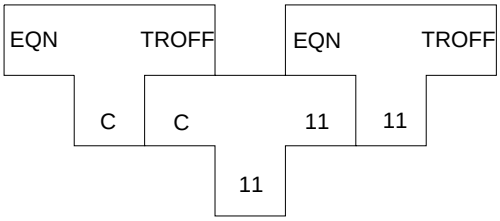


Fig. 14: Compilando el preprocesador EQN

Existen otros procesadores de texto comerciales, con preprocesadores para el tratamiento de expresiones matemáticas, tal es el caso del Lotus Manuscript® y T_EX®, que trabajan en el sistema operativo DOS. En el entorno WINDOWS® los procesadores Lotus AMIPRO® y Microsoft WORD® fueron los pioneros incorporando preprocesadores de fórmulas matemáticas, aunque actualmente la mayor parte de los procesadores de texto comerciales las incorporan.

5. BOOTSTRAPPING

El *bootstrapping* (literalmente "truco o trampa de arranque"[1]) se basa en utilizar las facilidades que ofrece un lenguaje para compilarse a sí mismo. El bootstrapping puede plantear la cuestión: ¿Qué fue primero, el huevo o la gallina?

Una forma de bootstrapping es construir el compilador de un lenguaje a partir de un subconjunto de dicho lenguaje. Supóngase que un nuevo lenguaje L se va a implementar en una máquina M. Como primer paso se puede escribir un pequeño compilador que traduce un subconjunto S de L a un lenguaje objeto que es el código de la máquina M; es decir un compilador S_MM. Entonces se puede usar el subconjunto S para escribir un compilador L_SM para L. Cuando L_SM se ejecuta a través de S_MM, se obtiene una implementación de L, que es L_MM. El lenguaje *Neliac* (un derivado del Algol) fue uno de los primeros lenguajes que se implementó a partir de su propio lenguaje [HUSK60, AHO86].

El lenguaje Pascal fue implementado por primera vez por *Niklaus Wirth (1971)* por medio de un compilador escrito en un subconjunto del Pascal (más del 60%). También escribió un compilador de este subconjunto en un lenguaje de bajo nivel, y por medio de varios booststrapping logró construir un compilador para el Pascal completo. Los métodos utilizados para realizar el booststrapping de los compiladores de Pascal se describen por *Lecarme y Peyrolle-Thomas* [LECA78].

Las mayores ventajas del bootstrapping se producen cuando un compilador se escribe en el lenguaje que se compila. Supóngase que escribimos un un compilador L_LN, para un lenguaje L, escrito en L para una máquina N. Pero las tareas de desarrollo se llevan a cabo en otra máquina M, donde existe un compilador L_MM. En un primer paso se obtiene un compilador cruzado L_MN, que se ejecuta sobre M y produce el código para N:

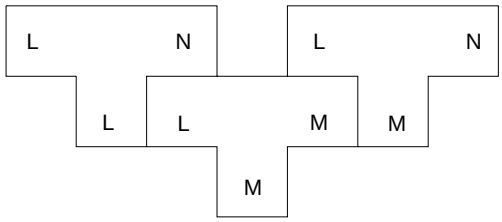


Fig. 15: Compilando el compilador L_LN

El compilador L_LN puede compilarse por segunda vez, pero ahora usando el generado por el compilador cruzado:

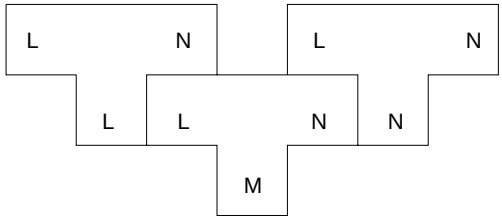


Fig. 16: Compilando L_LN con el obtenido anteriormente

El resultado de la segunda compilación es un compilador L_NN que se ejecuta en una máquina N, y produce un código para N. Sin embargo el compilador está construido en una máquina M (véase ejemplo 4.1). El proceso se muestra en la siguiente figura:

1 El término original en inglés *bootstrapping* se refiere a un antiguo refrán popular en lengua inglesa que se decia a los niños para que "crecieran tirando de sus propias pantorrillas".

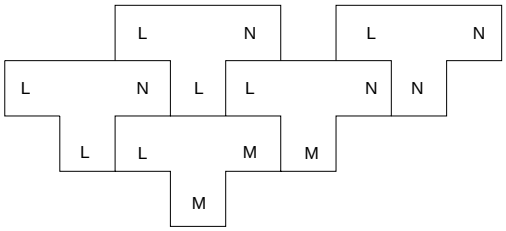


Fig. 17: Bootstrapping sobre un compilador

Utilizando las técnicas del bootstrapping, un compilador puede optimizarse a sí mismo. Supongamos todo el desarrollo en una máquina dada M. Tenemos $S_S M$, un buen compilador optimizado para el lenguaje S escrito en S, y queremos $S_M M$, un buen compilador optimizado escrito en M.

Se puede construir $S_{M*} M*$, un compilador poco optimizado de S en M, que no sólo genera mal código, sino que además es lento. (M* indica una mala implementación en M. $S_{M*} M*$ es una implementación peor del compilador que genera peor código). Sin embargo, se puede usar este compilador no optimizado, para obtener un buen compilador de S en dos pasos:

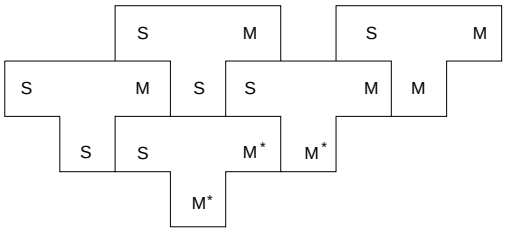


Fig. 18: Optimización de un compilador

En primer lugar, el compilador optimizado $S_S M$ se compila con $S_{M*} M*$, y produce $S_{M*} M$, que es un compilador lento pero que produce un código optimizado. El compilador optimizado $S_M M$, se obtiene al compilar $S_S M$ con $S_{M*} M$.

Ejemplo 5.1: Compilador de FORTRAN H

La técnica de bootstrapping se ha utilizado para el desarrollo del compilador FORTRAN H. El compilador se escribió en FORTRAN. El primer paso fue convertir la ejecución sobre un IBM 7094 a un System/360, una tarea dura. El segundo paso fue optimizarse, pasar de un tamaño de alrededor de 550K a unos 400K bytes [LOWR69, AHO86].

Ejemplo 5.2: Compilador de Pascal

Ammann [AMMA81] describe como se desarrolló el compilador de Pascal, diviendo la tarea en varias fases:

a) Se eligió un subconjunto básico de Pascal, y se programó en SCALLOP (lenguaje de nivel medio de los ordenadores CDC). Este primer conjunto se denominó **Pascal sin revisar PSR**. Por medio de un bootstrapping se consiguió un compilador $PSR_{6000} 6000*$, denotando por 6000, el código de las serie CDC 6000, y por 6000* el código no óptimo.

b) Se programó un nuevo compilador de **Pascal revisado P**, utilizando el Pascal sin revisar, $P_{PSR} 6000$.

c) Se compila el compilador obtenido en b) con el obtenido en a), y se obtiene $P_{6000*} 6000$

d) Se vuelve a programar un compilador de Pascal revisado, pero esta vez en Pascal revisado, $P_P 6000$.

e) Se compila el compilador d) con el obtenido en c) y se obtiene por fin $P_{6000} 6000$.

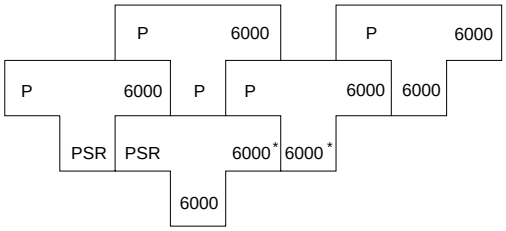


Fig. 19: Bootstrapping del compilador Pascal revisado

6. AUTOCOMPILADOR

Un *autocompilador de un lenguaje es un compilador que puede compilarse a sí mismo*. Es decir el lenguaje fuente y el lenguaje de implementación es el mismo, y el compilador es capaza de compilar el programa fuente del propio compilador. No debe confundirse autocompilación con bootstrapping, la autocompilación puede ser el resultado final de un bootstrapping, pero también puede llegarse a ella utilizando otro compilador del mismo lenguaje.

Hendrix [HEND88] relata como construir un autocompilador de *Small C* (un subconjunto del lenguaje C). El libro incluye un disquete con todos los fuentes y un compilador de *Small C*.

Otro ejemplo de autocompilador es el *Ccc32*, compilador de C concurrente para Windows [GOME97a, GOME97b].

7. LENGUAJES, GRAMÁTICAS Y AUTÓMATAS

Un lenguaje se puede definir, desde el punto de vista lingüístico, como un conjunto finito o infinito de oraciones, cada una de ellas de longitud finita y construidas por concatenación a partir de un número finito de elementos. Desde el punto de vista informático, un lenguaje es una notación formal para describir algoritmos o funciones que serán ejecutadas por un ordenador. No obstante la primera definición, debida a *Chomsky*, es válida para cualquier lenguaje sea natural o de programación.

El concepto de gramática fue acuñado por los lingüistas en sus estudios sobre los lenguajes naturales. Los objetivos de una gramática son:

- definir si una sentencia pertenece o no al lenguaje
- describir estructuralmente las sentencias.

En los lenguajes de programación los dos objetivos anteriores siguen vigentes.

Una gramática es un ente formal que especifica de una manera finita, un conjunto de sentencias, o cadenas de símbolos, potencialmente infinito (y que constituye un lenguaje).

Un **lenguaje** *es un conjunto de cadenas de símbolos de un alfabeto determinado*. Entendiéndose por símbolo una entidad abstracta, que se dejará como axioma, al igual que no se define punto en Geometría. Alfabeto o vocabulario es un conjunto finito de símbolos. Cadena es una secuencia finita de símbolos de un determinado alfabeto.

Una **gramática** se puede definir formalmente como *una cuadrupla formada por un vocabulario terminal VT, un vocabulario no terminal VN, un símbolo inicial S, y un conjunto de producciones o reglas de derivación P*.

$$G = (VT, VN, S, P)$$

Todas las sentencias del lenguaje definido por la gramática están formadas con símbolos del **vocabulario terminal** VT o **símbolos terminales**.

El **vocabulario no terminal** VN es un conjunto de símbolos introducidos como elementos auxiliares para la definición de las producciones de la gramática, y que no figuran en las sentencias del lenguaje.

Evidentemente, el conjunto de símbolos terminales y el conjunto de símbolos no terminales no tienen ningún elemento en común.

El **símbolo inicial** S es un símbolo perteneciente al vocabulario no terminal, y a partir del cual pueden obtenerse todas las sentencias del lenguaje generado por la gramática.

Las **producciones o reglas de derivación** P son transformaciones de cadenas de símbolos, que se expresan mediante una pareja de expresiones separadas por una flecha. A la izquierda de la flecha está el símbolo o conjunto de símbolos a transformar, y a la derecha se sitúan los símbolos resultado de la transformación.

Las reglas de derivación pueden aplicarse sucesivamente, desde el símbolo inicial hasta obtener una cadena de símbolos terminales, con lo que se obtiene una sentencia del lenguaje.

La definición formal de **lenguaje**, es el conjunto de todas las sentencias formadas por símbolos terminales que se pueden generar a partir de una gramática. Así si se denota L(G), al lenguaje generado por una gramática G, se puede expresar de la siguiente forma:

$$L(G) = \{\alpha \in VT / S \rightarrow \alpha\}$$

donde α representa una cadena de símbolos terminales.

Entonces se dice que una sentencia pertenece a un lenguaje L(G) si:

- está compuesta de símbolos terminales
- puede derivarse del símbolo inicial S, por medio de las distintas producciones de la gramática G.

Se dice que dos gramáticas son equivalentes, si ambas generan el mismo lenguaje.

La tarea de comprobar si una sentencia o instrucción pertenece o no a un determinado lenguaje se encomienda a los autómatas. La palabra autómata evoca algo que pretende imitar las funciones propias de los seres vivos, especialmente relacionadas con el movimiento, por ejemplo el típico robot antropomorfo. En el campo de los procesadores de lenguaje lo fundamental no es la simulación del movimiento, sino la simulación de procesos para tratar información.

La información se codifica en cadenas de símbolos, y un autómata es un dispositivo que se le presentan a su entrada, produciendo otras cadenas de símbolos de salida. El autómata recibe los símbolos de entrada, uno detrás de otro, es decir secuencialmente. El símbolo de salida que en un instante determinado produce un autómata, no sólo depende del último símbolo recibido a la entrada, sino de toda la secuencia o cadena de símbolos, recibida hasta ese instante. Todo lo anterior conduce a definir un concepto fundamental: **estado de un autómata**. *El estado de un autómata es toda la información necesaria en un momento dado, para poder deducir, dado un símbolo de entrada en ese momento, cual será el símbolo de salida*. Es decir, conocer el estado de un autómata, es lo mismo que conocer toda la historia de símbolos de entrada, así como el **estado inicial**, *estado en que se encontraba el autómata al recibir el primero de los símbolos de entrada*.

El autómata tendrá un determinado número de estados (pudiendo ser infinitos), y se encontrará en uno u otro según sea la historia de símbolos que le han llegado.

Se define **configuración** de un autómata a su situación en un instante. Si un autómata se encuentra en una configuración, y recibe un símbolo, producirá un símbolo de salida y efectuará un cambio o **transición** a otra configuración.

En el campo de estudio de los traductores, compiladores, procesadores e intérpretes los autómatas se utilizan como **reconocedores de lenguajes**, que dada una cadena de símbolos indican si dicha cadena pertenece o no al lenguaje.

Un autómata se puede definir como una quíntupla A=(E, Q, f, q_0, F), donde E es el *conjunto de entradas o vocabulario de entrada*, Q es el *conjunto de estados*, f : E×Q→Q es la *función de transición* o función del estado siguiente, q_0 es el *estado inicial*, F es el *conjunto de estados finales* o estados de aceptación.

Una cadena pertenece a un lenguaje si el autómata reconocedor de dicho lenguaje la toma como entrada, y partiendo del estado inicial transita a través de varias configuraciones hasta que alcanza el estado final.

Gramáticas	Lenguajes	Autómatas
No restringidas	Tipo 0	Máquinas de Turing
Sensibles al contexto	Tipo 1	A. Lineales Acotados
Libres de contexto	Tipo 2	Autómatas de Pila
Regulares	Tipo 3	Autómatas finitos

Tabla 1: Relaciones entre los distintos tipos gramáticas, lenguajes y autómatas.

En tabla 1 se presenta una clasificación de los lenguajes, gramáticas y autómatas, así como la relación entre ellas. La teoría de lenguajes, gramáticas y autómatas constituye una de las bases formales de la construcción de procesadores de lenguaje [CUEV91].

Ejemplo 7.1

Sea la gramática G = ({S}, {a,b}, P, S), donde P son las reglas de producción:

$$S \rightarrow a\ b$$
$$S \rightarrow a\ S\ b$$

La única forma de generar sentencias con esta gramática es aplicando cualquier número de veces la segunda producción, y terminando con la aplicación de la primera.

$S \rightarrow a\ b$
$S \rightarrow a\ S\ b \rightarrow a\ a\ b\ b$
$S \rightarrow a\ S\ b \rightarrow a\ a\ S\ b\ b \rightarrow a\ a\ a\ b\ b\ b$
. . .
$S \rightarrow a\ S\ b \rightarrow \cdots \rightarrow a^n\ b^n$

El lenguaje generado es $L(G)=\{a^n\ b^n / n \geq 1\}$

Ejemplo 7.2

Sea la gramática G = ({S,A,B}, {a,b}, S, P), donde P son las reglas de producción:

$$S \rightarrow a\ B$$
$$S \rightarrow b\ A$$
$$A \rightarrow a$$
$$A \rightarrow a\ S$$
$$A \rightarrow b\ A\ A$$
$$B \rightarrow b$$
$$B \rightarrow b\ S$$
$$B \rightarrow a\ B\ B$$

Hopcroft J.E. y J. D. Ullman [HOPC79], pp. 81-82 demuestran que el lenguaje generado son las cadenas con igual número de símbolos *a* que de símbolos *b*.

Ejemplo 7.3

Se desea construir una gramática que describe el lenguaje formado por los identificadores de un lenguaje de programación (cadenas que comienzan por una letra, que puede ser seguida por letras o dígitos), y diseñar un autómata que reconozca identificadores. La gramática es la siguiente:

```
S → <IDENT>
<IDENT> → letra
<IDENT> → <IDENT> letra
<IDENT> → <IDENT> dígito
```

El autómata A=(E,Q,f,q_0,F) se representa por medio de un diagrama de Moore en la fig. 20, donde E={letra,dígito}, Q={S,<IDENT>}, q_0=S, F={<IDENT>}, y la función de transición viene es la representada en tabla 2.

f	letra	dígito
S	<IDENT>	-
<IDENT>	<IDENT>	<IDENT>

Tabla 2: Función de transición del autómata

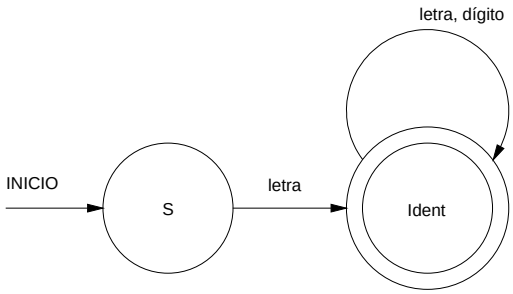


Fig. 20: Autómata reconocedor de identificadores

8. LÉXICO

El léxico de un lenguaje natural está constituido por todas las palabras y símbolos que lo componen. Para un lenguaje de programación u otro lenguaje usado en Informática la definición anterior también es válida.

En los lenguajes de programación el léxico lo constituyen todos los elementos individuales del lenguaje, denominados frecuentemente en inglés *tokens*. Así son tokens: las palabras reservadas del lenguaje, los símbolos que denotan los distintos tipos de operadores, identificadores (de variables, de funciones, de procedimientos, de tipos, etc...), separadores de sentencias y otros símbolos empleados en las distintas construcciones de un lenguaje.

9. SINTAXIS

En lingüística, sintaxis es el estudio de la función que desempeña cada palabra en el entorno de una frase. Mientras que semántica es el estudio del significado de una palabra tanto a nivel individual como en el contexto de una frase.

En los lenguajes de programación, sintaxis es un conjunto de reglas formales que especifican la composición de los programas a base de letras, dígitos y otros caracteres. Por ejemplo, las reglas de sintaxis especifican en Pascal que dos sentencias deben ir separadas por un ";" o que la declaración de tipos debe ir antes que la de variables.

10. SEMÁNTICA

La semántica en lingüística es el estudio del significado de una palabra tanto a nivel individual como en el contexto de una frase.

Semántica en los lenguajes de programación es el conjunto de reglas que especifican el significado de cualquier sentencia, sintácticamente correcta y escrita en un determinado lenguaje. Por ejemplo en el lenguaje Pascal la sentencia :

```
suma := 27 / lado
```

es sintácticamente correcta, ya que a la izquierda del símbolo de asignación hay un identificador, y a la derecha una expresión. Pero para que sea semánticamente correcta hay que comprobar:

- *lado* debe ser compatible con el operador "/" y con el operando 27.
- *suma* debe ser un tipo compatible con el resultado de la operación.

11. METALENGUAJES

Los lenguajes de programación son un conjunto finito o infinito de sentencias. Los lenguajes de programación con un número de sentencias finito se pueden especificar exhaustivamente enumerando todas sus sentencias. Sin embargo, para los lenguajes con un número de sentencias infinito ésto no es posible, pues los medios que tenemos para describirlo son finitos. El medio habitual para describir un lenguaje es su **gramática**, pero las gramáticas que se utilizan en los lenguajes de programación necesitan una modelización metamatemática que haga factible su implementación en los compiladores. En los años 50 *Noam Chomsky* realizó grandes avances en la concepción de un modelo matemático de las gramáticas en conexión con los lenguajes naturales.

Los **metalenguajes** *son herramientas para la descripción formal de los lenguajes*, facilitando no sólo la comprensión del lenguaje, sino también el desarrollo del compilador correspondiente. Ejemplos: las expresiones regulares que describen los componentes léxicos del lenguaje; las notaciones BNF, EBNF y los diagramas sintácticos que describen la sintaxis de los lenguajes.

Las características semánticas de los lenguajes pueden describirse en lenguaje natural, pero éste es demasiado impreciso, por lo que se utilizan especificaciones formales, entre los cuales se pueden destacar [CAST93 (pp.146), PARE94 (pp.137)] *semática operacional*, *semántica denotacional*, *semántica axiomática* y semántica basada en *gramáticas con atributos*.

Las gramáticas con atributos o atribuidas son una extensión de la notación BNF que incluyen aspectos semánticos. Existen también otros metalenguajes orientados a la definición de las características semánticas, basados habitualmente en lenguajes de especificación formal o herramientas de ingeniería del software por ejemplo las notaciones VDM (*Vienna Development Method*) [DAWE91, HEKM88, SANC89], HDM (*Hierarchical Development Method*) [LEVI79, SANC89] y ANNA (*ANNotated Ada*) [KRIE83, SANC89].

11.1 Expresiones regulares

Las expresiones regulares es un metalenguaje para describir los elementos léxicos de un lenguaje o *tokens* [AHO86, CUEV91, CUEV93a]. Las expresiones regulares utilizan cuatro tipos de operadores para definir los componentes léxicos, que se definen a continuación de mayor a menor precedencia:

- *Paréntesis*. Se permite el uso de paréntesis para agrupar símbolos.
- *Operaciones de cierre y cierre positivo*. Se define la operación de cierre u operación estrella sobre una cadena α, es decir α^*, como el conjunto de cadenas: λ, α, $\alpha\alpha$, $\alpha\alpha\alpha$, $\alpha\alpha\alpha\alpha$, ..., $\alpha\alpha\alpha...\alpha$. Es decir representa: la cadena vacia λ, la propia cadena, una repetición, dos, etc... un valor indefinido de repeticiones.
  El cierre positivo sobre una cadena α, es decir α^+, como el conjunto de cadenas: α, $\alpha\alpha$, $\alpha\alpha\alpha$, $\alpha\alpha\alpha\alpha$, ..., $\alpha\alpha\alpha...\alpha$. Es decir representa: la propia cadena, una repetición, dos, etc... un valor indefinido de repeticiones pero sin incluir la cadena vacia.
- *Operación concatenación*. Se permite la concatenación de cadenas.
- *Operación alternativa.* Se representa por |, y permite la elección entre dos alternativas.

Ejemplo 11.1.1: Identificador

Un identificador se puede definir a partir de dos símbolos: *letra* y *dígito*. Así un identificador se puede definir con la expresión regular:

$$letra\ (\ letra \mid dígito\ )^*$$

Ejemplo 11.1.2: Constante entera

Una constante entera sin signo se puede definir como: $dígito^*$

Con signo se puede definir como: $(\lambda \mid - \mid +\ )\ dígito^*$

Ejemplo 11.1.3: Constante de punto flotante sin exponente

Se puede definir como: $(\lambda \mid - \mid +\ )\ dígito^*.dígito^+$

11.2 Diagramas Sintácticos

Los diagramas sintácticos son metalenguajes para la descripción sintáctica de lenguajes de programación (por ejemplo el lenguaje Pascal). Constan de una serie de cajas o símbolos geométricos conectados por arcos dirigidos.

Los **símbolos terminales** se introducen dentro de círculos o cajas de bordes redondeados (fig. 21).

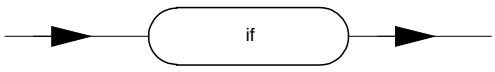


Fig. 21: Diagrama sintáctico de un símbolo terminal

Los **símbolos no terminales** se representan por su nombre encerrado en un rectángulo o cuadrado (fig. 22).

Fig. 22: Diagrama sintáctico de un símbolo no terminal

Las reglas sintácticas con varias alternativas se representan en la fig. 23.

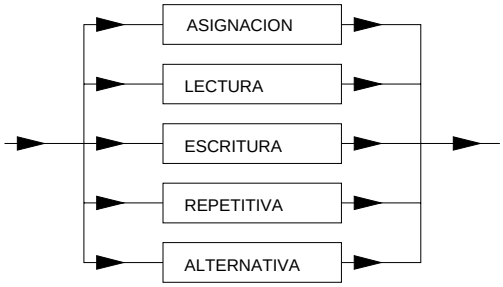


Fig. 23: Diagrama sintáctico de reglas con varias alternativas

Las contrucciones con cero o más repeticiones de símbolos se representan en la fig. 24.

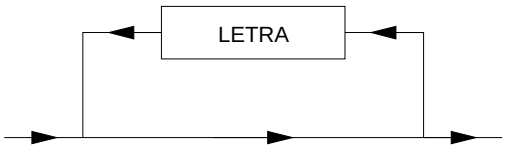


Fig. 24: Diagrama sintáctico de construcciones repetitivas

La opcionalidad de un no terminal o terminal se representa en la fig. 25.

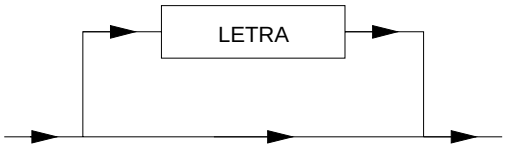
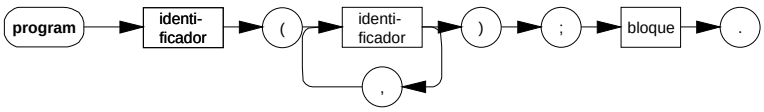


Fig. 25: Diagrama sintáctico de una regla con una alternativa vacia

Ejemplo 11.2.1: Diagramas sintácticos del lenguaje Pascal

El esquema general del lenguaje Pascal puede observarse en los diagramas de la figura 26, el resto se muestra en el anexo III.1 según la norma ISO Pascal, *Jensen K. y N. Wirth* [JENS75].



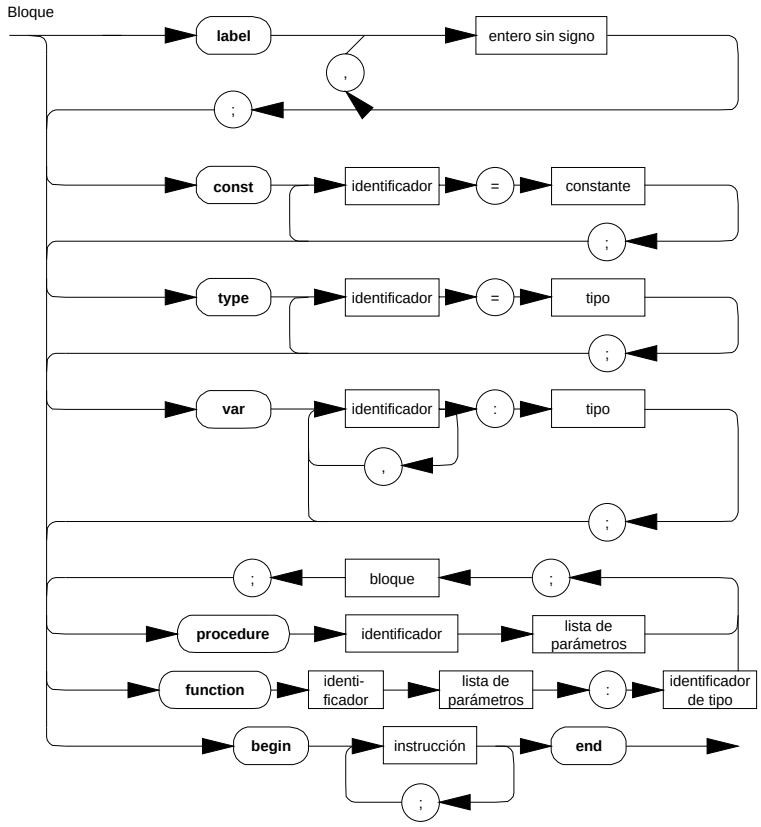


Fig. 26: Diagramas sintácticos genéricos de un programa en Pascal

11.3 La notación BNF (Backus-Naur Form)

John Backus participó en la creación del primer lenguaje de alto nivel, el lenguaje FORTRAN, y posteriormente participó a principios de los años 60 en el desarrollo del lenguaje ALGOL, que utilizó por primera vez la forma BNF.

La notación BNF utiliza los siguientes metasímbolos:

< >	encierra conceptos definidos o por definir. Se utiliza para los símbolos no terminales.
::=	sirve para definir o indicar equivalencia
|	separa las distintas alternativas
" "	Indica que el metasímbolo que aparece entre comillas es un carácter que forma parte de la sintaxis del lenguaje
( )	Se permite el uso del paréntesis para hacer agrupaciones

Existen símbolos con entidad propia llamados símbolos terminales. También existen otros que se deben de definir y se denominan no terminales.

Ejemplo 11.3.1: Definición de un identificador

La definición de un identificador en BNF es la siguiente, véase que se permiten definiciones recursivas.

```
<ident>      ::= <letra> | <ident> <letra> | <ident> <digito>
<letra>      ::= a | b | c | ... | y | z
<digito>     ::= 0 | 1 | 2 | ... | 8 | 9
```

En este caso a, b, c, ..., y, z y 0, 1, 2,...9 son los símbolos terminales, y el resto los símbolos no terminales. Puede observarse que se ha definido identificador de una manera recursiva.

11.4 Notación EBNF

La notación EBNF (*Extended BNF*), añade el metasímbolo { } a la notación BNF, para indicar que lo que aparece entre las llaves puede repetirse cero o más veces. En algunos casos se indica con subíndices y superíndices el intervalo de repeticiones.

Ejemplo 11.4.1: Definición de identificador en FORTRAN

En el lenguaje FORTRAN 77 un identificador viene definido por

```
<idenFORTRAN>       ::= <letra> { <alfanumérico> }[0..5]

<alfanumérico>      ::= <letra> | <dígito>
<letra>             ::= a | b | c | ... | y | z
<dígito>            ::= 0 | 1 | 2 | ... | 8 | 9
```

Ejemplo 11.4.2: Expresión aritmética con uno o varios factores

Un término de una expresión aritmética se puede expresar en BNF como:

```
<term> ::= <factor> | <term> * <factor>
```

sin embargo usando EBNF se puede expresar como:

```
<term> ::= <factor> { * <factor> }
```

Ejemplo 11.4.3: Gramática del lenguaje Pascal

La gramática del lenguaje Pascal en EBNF, *Jensen K. y N. Wirth* [JENS75]. Se muestra en el Anexo III.2.

11.5 Gramáticas atribuidas o gramáticas con atributos

Las gramáticas con atributos o atribuidas, fueron definidas originalmente por *Knuth (1968)* para definir las especificaciones semánticas de los lenguajes de programación. Una gramática atribuida **se caracteriza** por [CUEV95b]:

1.- La estructura sintáctica del lenguaje se define mediante una gramática libre de contexto.

2.- Cada símbolo de la gramática, tiene asociado un conjunto finito de atributos. Ejemplo:

<SIMBOLO>.a.b.c

siendo *a*, *b* y *c* atributos.

Cada atributo puede tomar un conjunto de valores (posiblemente infinito). El valor de un atributo describe una propiedad dependiente del contexto del símbolo en cuestión

3.- Cada regla de producción debe de especificar como se modifican los atributos con su aplicación.

4.- Una gramática atribuida describe un sublenguaje (del lenguaje definido) mediante las condiciones de contexto que deben ser cumplidas por los valores de los atributos. Una sentencia sintácticamente correcta también lo será semánticamente si y sólo si todos los atributos satisfacen las condiciones del contexto.

El principal **problema** del proceso de gramáticas atribuidas, es que el tiempo necesario para realizar las comprobaciones semánticas *crece exponencialmente con el número de atributos*. La evaluación de los atributos se realiza mediante las reglas semánticas. Los atributos se pueden clasificar en *heredados* y *sintetizados*.

Dada una gramática con atributos, se dice que un atributo *a* asociado a un símbolo X es *heredado* si existe una regla sintáctica de la forma $Y \rightarrow \alpha X \beta$ y una regla semántica que calcula *a* con los atributos del resto de los símbolos que forman la regla.

Dada una gramática con atributos, se dice que un atributo *a* asociado a un símbolo X es *sintetizado* si existe una regla sintáctica de la forma $X \rightarrow \alpha$ y una regla semántica que calcula *a* con los atributos de los símbolos de la parte derecha de dicha regla.

Ejemplo 11.5.1: gramática con atributos de un lenguaje con expresiones aritméticas

Sea una pequeña gramática libre de contexto para expecificar expresiones aritméticas y asignaciones de un intérprete.

(1) <ASIGNACION> → <VARIABLE> = <EXPRESION>

(2) <EXPRESION> → <EXPRESION> <OPERADOR> <EXPRESION>

(3) <EXPRESION> → **número**

(4) <VARIABLE> → **identificador**

(5) <OPERADOR> → +

(6) <OPERADOR> → -

Ejemplos de sentencias de este lenguaje son: *c = 1+1*, *velocidad = 3.2 + 4 - 5.0*, *tocino = 4 - 5.1*.

Se pueden definir los siguientes atributos:

número.tipo	Tipo del número (entero o real)
número.valor	Valor que contiene el *token* número.
<VARIABLE>.tipo	Tipo de la variable

<VARIABLE>.valor	Valor que toma la variable
<EXPRESION>.tipo	Tipo de la expresión
<EXPRESION>.valor	Valor de la expresión después de evaluarla
<OPERADOR>.tipo	Tipo del operador (entero o real)
<OPERADOR>.clase	Clase del operador (adición y sustracción)

El símbolo terminal **identificador** no tiene el atributo tipo en este ejemplo dado que no existe declaración obligatoria de variables en esta gramática. En el caso de una gramática con declaraciones obligatorias, en la zona de declaraciones es donde los identificadores reciben el atributo tipo.

Continuando con la gramática del ejemplo, se puede ampliar la gramática dada con las siguientes acciones y reglas semánticas para construir un intérprete puro del lenguaje definido por dicha gramática (ver apartado 13.1 para la definición de intérprete puro). Puede observarse que en la especificación semántica se propaga el tipo y el valor de las expresiones, varibles y operadores. Cuando un símbolo se repite se le coloca un subíndice de derecha a izquierda para que las reglas y acciones semánticas no sean ambiguas.

```
(1)  <ASIGNACION> → <VARIABLE> = <EXPRESION>
          {<VARIABLE>.valor := <EXPRESION>.valor
           <VARIABLE>.tipo := <EXPRESION>.tipo}


(2)  <EXPRESION> → <EXPRESION> <OPERADOR> <EXPRESION>
          { <OPERADOR>.tipo := mayor_tipo(<EXPRESION>2.tipo, <EXPRESION>3.tipo)
            <EXPRESION>1.tipo := <OPERADOR>.tipo

            if (<OPERADOR>.tipo=='F' && <EXPRESION>2.tipo=='I')
                     {<EXPRESION>2.tipo:='F';
                      <EXPRESION>2.valor:=FLOAT(<EXPRESION>2.valor);}

            if (<OPERADOR>.tipo=='F' && <EXPRESION>3.tipo=='I')
                     {<EXPRESION>3.tipo:='F';
                      <EXPRESION>3.valor:=FLOAT(<EXPRESION>3.valor);}

            switch (<OPERADOR>.tipo)
                    {
               'I': <EXPRESION>1.valor :=op_entera(
                        <OPERADOR>.clase, <EXPRESION>2.valor, <EXPRESION>3.valor); break;
               'F': <EXPRESION>1.valor :=op_real(
                        <OPERADOR>.clase, <EXPRESION>2.valor, <EXPRESION>3.valor); break;
                    }
          }


(3)  <EXPRESION> → número
          { <EXPRESION>.valor := número.valor
            <EXPRESION>.tipo := número.tipo
          }


(4)  <VARIABLE> → identificador


(5)  <OPERADOR> → + {<OPERADOR>.clase:='+'}


(6)  <OPERADOR> → - {<OPERADOR>.clase:='-'}
```

Las funciones auxiliares utilizadas en las reglas semánticas se definen a continuación.

```
tipo Mayor_Tipo (char tipo1, char tipo2)
 {
  if (tipo1==tipo2) return tipo1;
  if (tipo1=='F' || tipo2=='F') return 'F';
    else return 'I';
 }
```

```
int op_entera(char op, int valor1, int valor2)
 {
  switch (op)
   {
     '+': return (valor1 + valor2);
     '-': return (valor1 - valor2);
   }
float op_real(char op, float valor1, float valor2)
 {
  switch (op)
   {
     '+': return (valor1 + valor2);
     '-': return (valor1 - valor2);
   }
```

Las función *FLOAT* convierte un entero en real.

Se utilizarán las reglas semánticas para evaluar los atributos de la sentencia:

```
velocidad = 80.5 + 20
```

Se construye el árbol sintáctico con los atributos que se muestra en la figura 27.

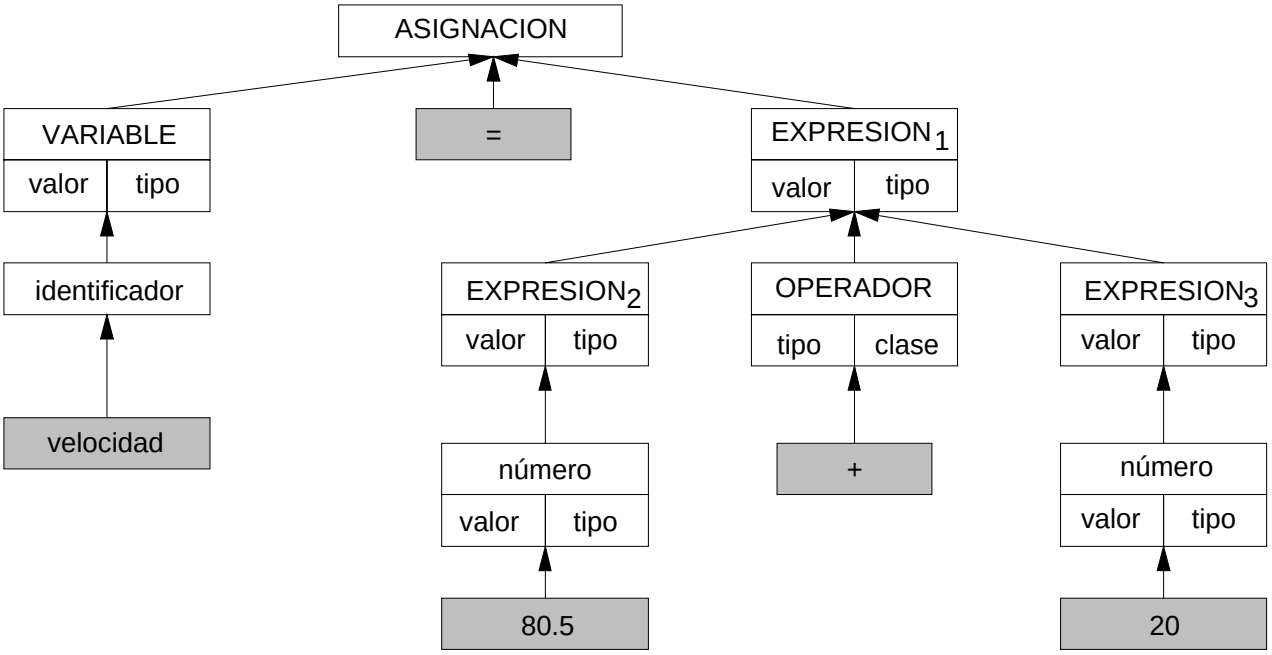


Figura 27: Arbol sintáctico con atributos

El analizador léxico devuelve los *tokens* reconocidos con sus atributos:

```
velocidad.identificador = 80.5.float + 20.int
```

A partir de esta información los atributos se propagan según las reglas y acciones semánticas, obteniéndose:

<EXPRESION>$_2$.valor:=80.5
<EXPRESION>$_2$.tipo:='F'
<EXPRESION>$_3$.valor:=20
<EXPRESION>$_3$.tipo:='I'
<OPERADOR>.clase:='+'
<OPERADOR>.tipo:='F'
<EXPRESION>$_1$.tipo:='F'
<EXPRESION>$_3$.tipo:='F'
<EXPRESION>$_3$.valor:=20.0
<EXPRESION>$_1$.valor:=100.5
<VARIABLE>.tipo:='F'
<VARIABLE>.valor:=100.5

12. ESTRUCTURA GENERAL DE UN TRADUCTOR

La construcción de un traductor de un determinado lenguaje es una tarea compleja, que se puede reducir siguiendo una metodología, que consistirá en dividir en módulos las distintas fases del traductor. La **complejidad** del traductor dependerá de las características del lenguaje fuente y del lenguaje objeto, y de su distancia medida en términos de diferencias de niveles. Así es más sencillo traducir entre dos lenguajes del mismo nivel, que entre un lenguaje de alto nivel y uno de bajo nivel.

En la figura 28 se muestra la estructura de un traductor, que debemos situarla en el contexto del módulo traductor de la figura 8. En un principio todas las fases se pueden agrupar en dos tareas: el análisis del programa fuente y su síntesis en el correspondiente programa objeto.

El **análisis** consiste en verificar la corrección del programa fuente, para lo cual se descompone el programa fuente en trozos elementales o unidades mínimas sintácticas denominadas *componentes léxicos* o en inglés *tokens* (análisis léxico). Los *tokens* se pueden agrupar para comprobar su disposición correcta en las distintas construcciones y sentencias del lenguaje a analizar (análisis sintáctico y semántico). Comprobándose de esta forma la validez sintáctica y semántica del programa fuente. En caso contrario se emiten los errores oportunos (manejo de errores). La información de cada identificador recogida durante la fase de análisis se almacena en la tabla de símbolos. Esta información también se utiliza durante la fase de análisis (por ejemplo para comprobar si una variable se intento declarar dos veces).

La tarea de **síntesis** tiene por objeto la generación del código del lenguaje objeto. En el caso particular de los compiladores suele incluirse también la generación de código intermedio, como un medio para garantizar la transportabilidad entre distintas máquinas objeto o como método para poder utilizar el mismo back-end, entre compiladores de lenguajes diferentes.

La generación de código intermedia se apoya directamente en la información recogida en la tabla de símbolos durante la fase de análisis. La generación de código para la máquina objeto definitiva tan sólo se apoya en el código intermedio. También es necesario un tratamiento de errores para la fase de síntesis.

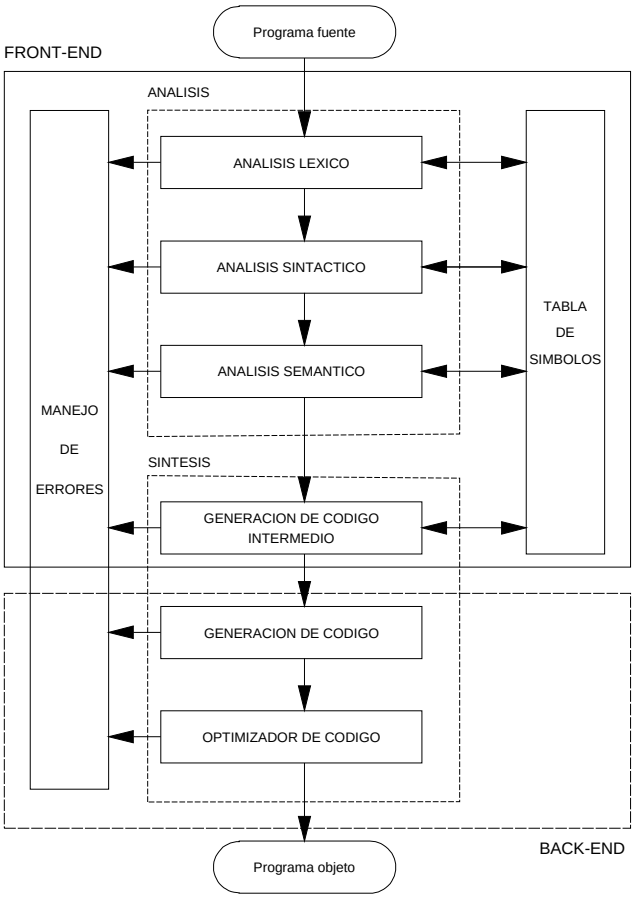


Fig. 28: Fases de un traductor

12.1 Análisis léxico

Un programa fuente es una serie de símbolos (letras, dígitos, símbolos especiales: +,*,&,$,#,!). Con estos símbolos se representan las construcciones del lenguaje tales como variables, etiquetas, palabras reservadas, constantes, operadores,... Es necesario que el traductor identifique los distintos significados de estas construcciones, que los creadores de lenguajes dan en la definición del lenguaje.

El programa fuente se trata inicialmente con el **analizador léxico** (en inglés *scanner*) con el proposito de agrupar el texto en grupos de caracteres con entidad propia llamados ***tokens*, unidades sintácticas o componentes léxicos**, tales como constantes, identificadores (de variables, de funciones, de procedimientos, de tipos, de clases), palabras reservadas, y operadores. Por razones de eficiencia, a cada token se le asocia un atributo (o más de uno) que se representa internamente por un código numérico o por un tipo enumerado. Por ejemplo a un identificador se le puede dar una representación interna de un 1, a una constante entera un 2, a un operador aritmético un 3,..., cada palabra reservada tiene su propio código. Así la siguiente sentencia de Pascal:

```
IF cuenta = sueldo THEN jefe := justo ;
```

el analizador léxico la separa en la siguiente secuencia de tokens:

y les asigna su atributo, habitualmente por medio de un código numérico cuyo significado se ha definido previamente.

token	Atributo	Observaciones
IF	20	Palabra reservada
cuenta	1	Identificador
=	15	Operador de comparación
sueldo	1	Identificador
THEN	21	Palabra reservada
jefe	1	Identificador
:=	10	Asignación
justo	1	Identificador
;	27	Separador de sentencias

El **análisis léxico** es un análisis a nivel de caracteres, su misión es reconocer los componentes léxicos o *tokens*, enviando al analizador sintáctico los *tokens* y sus atributos. También se encarga de eliminar los comentarios. El analizador léxico tambien recibe el nombre de **explorador** (en inglés *scanner*) [AHO86, HOLU90, CUEV93a].

Una herramienta para generar analizadores léxicos a partir de la definición de los componentes léxicos o *tokens* de un lenguaje es *lex*, originario inicialmente del sistema operativo UNIX, pero del que existen actualmente versiones prácticamente para todos los sistemas operativos, y algunas de ellas de dominio público *FLEX*, y otras desarrolladas por universidades (por ejemplo: *GALEX* en la Universidad de Oviedo [MART93]). Sobre el uso de estas herramientas para construir procesadores de lenguaje pueden consultarse distintos libros de texto [SCHR85, PYST88, HOLU90, BENN90, LEVI92, CUEV93a].

12.2 Análisis sintáctico

El **análisis sintáctico** (en inglés *parser*) es un análisis a nivel de sentencias, y es mucho más complejo que el análisis léxico. Su función es tomar el programa fuente en forma de *tokens*, que recibe del analizador léxico y determinar la estructura de las sentencias del programa. Este proceso es análogo a determinar la estructura de una frase en Castellano, determinando quien es el sujeto, el predicado, el verbo y los complementos. El análisis sintáctico agrupa a los *tokens* en clases sintácticas (denominadas no terminales en la definición de la gramática), tales como expresiones, procedimientos,... El analizador sintáctico o *parser* obtiene un **árbol sintáctico** (u otra estructura equivalente) en la cual las hojas son los *tokens* y cualquier nodo, que no sea una hoja, representa un tipo de clase sintáctica. Por ejemplo el análisis sintáctico de la siguiente expresión:

(A+B)*(C+D)

con las reglas de la gramática que se presenta a continuación dará lugar al **árbol sintáctico** de la figura 29.

```
<expresión>             ::= <término> <más terminos>
<más términos>          ::= + <término> <más términos>
                            | - <término> <más términos>
                            | <vacío>
<término>               ::= <factor> <más factores>
<más factores>          ::= * <factor> <más factores>
                            | / <factor> <más factores>
                            | <vacío>
```

```
<factor>                ::= ( <expresión> ) | <variable> | <constante>
```

La estructura de la gramática anterior refleja la prioridad de los operadores, así los operadores "+" y "-" tienen la prioridad más baja, mientras que "*" y "/" tienen una prioridad superior. Se evaluarán en primer lugar las constantes, variables y las expresiones entre paréntesis.

Los árboles sintácticos se construyen con un conjunto de reglas conocidas como **gramática**, y que definen con total precisión el lenguaje fuente.

Al proceso de reconocer la estructura del lenguaje fuente se le conoce con el nombre de **análisis sintáctico** (*parsing*). Hay distintas clases de analizadores o reconocedores sintácticos, en general se clasifican en dos grandes grupos: analizadores sintácticos ascendentes y analizadores sintácticos descendentes [AHO86, HOLU90, CUEV95a]. Véase Ejercicio 19.1: Compilador recursivo descendente de MUSIM/0.

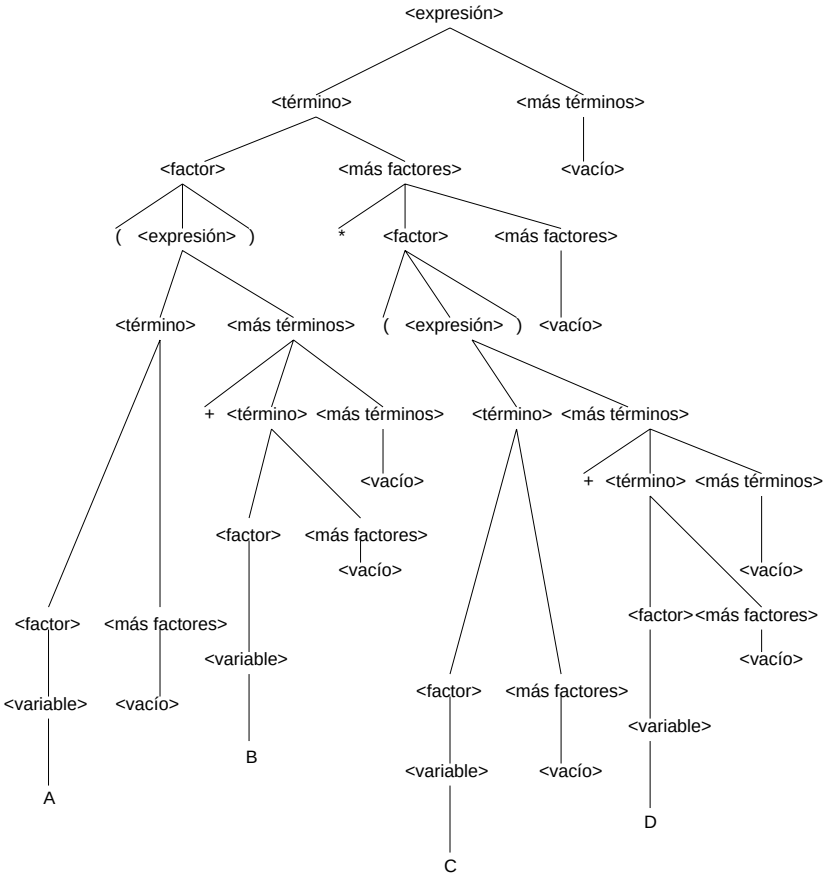


Fig. 29: Arbol sintáctico

12.2.1 Análisis sintáctico descendente

Los **analizadores sintácticos descendentes** construyen el árbol sintáctico a partir del símbolo inicial de la gramática, hasta llegar a los distintos tokens, que constituyen la sentencia a analizar. Es decir, se parte del símbolo inicial de la gramática y se van aplicando las distintas producciones hasta llegar a formar la sentencia. En el ejemplo de la figura 29, el árbol sintáctico se obtiene a partir de las reglas de derivación de <expresión> hasta llegar a la expresión. El orden de derivación es importante, siempre se deriva primero el no terminal situado más a la izquierda según se mira el árbol (**derivaciones más a la izquierda**).

Los principales problemas que se plantean en el análisis sintáctico descendente son dos: el retroceso (en inglés *backtracking*) y la recursividad a izquierdas.

El **retroceso**, se produce cuando se eligen mal las producciones para alcanzar la sentencia a analizar, y debe de darse marcha atrás en el proceso de análisis, para elegir otra alternativa. El retroceso se repite hasta que se ha elegido la alternativa correcta; evidentemente esto ralentiza enormemente el proceso de análisis.

La **recursividad a izquierdas**, se deriva de que en general se elige el criterio de tomar siempre en primera opción la alternativa más a la izquierda. Si en la gramática existen producciones de la forma: $A \rightarrow A\ \alpha$ entonces el analizador sintáctico se introduce en un bucle infinito, del que no puede salir.

Para evitar la recursividad a izquierdas y el retroceso, las gramáticas deben de cumplir unas determinadas propiedades, dando lugar a una clasificación de las gramáticas. Los lenguajes de programación que se pueden describir mediante uno de estos tipos de gramáticas, llamadas LL(k), se pueden analizar en forma descendente sin retroceso, y sin recursividad a izquierdas. Las gramáticas **LL(k)** realizan el análisis descendente determinista, por medio del reconocimiento de la cadena de entrada de izquierda a derecha (***Left to right***) y que va tomando las derivaciones más a la izquierda (***Leftmost***) con sólo mirar los **k** tokens situados a continuación de donde se halla. En general se utilizan las gramáticas LL(1) para la descripción de lenguajes, que sólo miran un token.

A continuación se muestra la traza de un analizador sintáctico recursivo descendente con el fragmento de gramática LL(1) mostrado anteriormente y que da lugar al árbol sintáctico de la fig. 29. Puede observarse como el analizador sigue el árbol sintáctico de la raíz a las hojas, eligiendo siempre la primera alternativa por la izquierda. El analizador léxico va leyendo los tokens y enviándoselos al analizador sintáctico. En el caso de que el token recibido no concuerde con el que contiene la regla, se devuelve al analizador léxico (caso de las derivaciones a <vacio>). Véase Ejercicio 19.1: Compilador recursivo descendente de MUSIM/0.

```
ANALISIS SINTACTICO: <EXPRESION>
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token (
ANALISIS SINTACTICO: <EXPRESION>
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token a
ANALIZADOR LEXICO: Recibe en buffer el token a
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token a
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token +
ANALIZADOR LEXICO: Recibe en buffer el token +
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token +
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token b
ANALIZADOR LEXICO: Recibe en buffer el token b
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token b
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ANALIZADOR LEXICO: Lee el token )
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token *
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token (
ANALISIS SINTACTICO: <EXPRESION>
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token c
ANALIZADOR LEXICO: Recibe en buffer el token c
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token c
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token +
ANALIZADOR LEXICO: Recibe en buffer el token +
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token +
ANALISIS SINTACTICO: <TERMINO>
```

```
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token d
ANALIZADOR LEXICO: Recibe en buffer el token d
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token d
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ANALIZADOR LEXICO: Lee el token )
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token ;
ANALIZADOR LEXICO: Recibe en buffer el token ;
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token ;
ANALIZADOR LEXICO: Recibe en buffer el token ;
```

También existe una herramienta denominada *Jack* para generar analizadores sintácticos descendentes en lenguaje Java [JACKi], a partir de unas especificaciones gramaticales del estilo de las herramientas clásicas de análisis sintáctico ascendente *lex* y *yacc*. Se caracteriza por:

- Genera código en lenguaje Java®
- Las especificaciones lexicas, sintácticas y semánticas van en un fichero único
- Permite el uso de cualquier carácter Unicode.
- Realiza análisis descendente utilizando una gramática LL(k), donde se puede fijar K, es decir el número de tokens que tiene que mirar hacia adelante (*lookahead*) para determinar la regla de la gramática que aplica.
- En el análisis semántico utiliza tanto atributos sintetizados como heredados (véase apartado 11.5).

12.2.2 Análisis sintáctico ascendente

Los **analizadores sintácticos ascendentes** construyen el árbol sintáctico a partir de las hojas, paso a paso hasta llegar a la raíz. Es decir parten de los distintos tokens de la sentencia a analizar, y por medio de **reducciones** llegan al símbolo inicial de la gramática. Se llaman reducciones, para indicar que se efectuan en sentido contrario a las producciones de la gramática. El principal problema que se plantea en el análisis ascendente es el del retroceso, que se traduce en la elección de un pivote para realizar la reducción. Para salvar este inconveniente se definieron distintos tipos de gramáticas entre las cuales las más utilizadas son las LR(k). Los analizadores sintácticos ascendentes que reconocen lenguajes descritos por **gramáticas LR(k)**, toman sus decisiones en función de los **k** tokens inspeccionados por delante (*lookaheads*) de cada vez en la cadena de entrada, tomados de izquierda a derecha (*Left to right*), realizándose el análisis por derivaciones más a la derecha en sentido inverso (*Rightmost)*. Las gramáticas LR(k) describen la mayor parte de los lenguajes libres de contexto, y son mucho menos restrictivas que las gramáticas LL(k). Sin embargo la complejidad de los analizadores sintácticos LR(k), hace que la mayor parte de estos se construyan por medio de herramientas software como por ejemplo el *yacc* incluido en el sistema operativo UNIX [KERN84].

También existen versiones de *yacc* para distintintos sistemas operativos, y de dominio público como *BISON*, o desarrollados por universidades (por ejemplo: *YACCOV*, *YACC de la universidad de OViedo* [CABA91]). A continuación se muestra un fragmento de gramática de entrada al *yacc* para reconocer la expresión (A+B)*(C+D). Puede observarse que se definen las precedencias y asociatividades de los operadores en primer lugar los menos prioritarios (*left* significa asociativo por la izquierda). Las reglas de la gramática se escriben siguiendo una notación parecida a la BNF, en la cual los símbolos terminales se escriben con mayúsculas (tokens enviados por el analizador léxico) y con minúsculas los símbolos no terminales. En el caso de símbolos terminales de un sólo carácter como los operadores aritméticos pueden escribirse directamente entre apóstrofos.

```
...
%left '-' '+'
%left '*' '/'
...
exp         : NUM               {  acciones semánticas y
                                   generación de código
                                   si fuese necesario    }
            | VAR               {  ... }
            | exp '+' exp       {  ... }
```

```
                        | exp '-' exp        {  ... }
                        | exp '*' exp        {  ... }
                        | exp '/' exp        {  ... }
                        | '(' exp ')'
                        ;
            ...
```

Sobre el uso de estas herramientas para construir procesadores de lenguaje pueden consultarse distintos libros de texto [SCHR85, PYST88, HOLU90, BENN90, LEVI92, CUEV95a]. Véase Ejercicio 19.6: Compilador ascendente de MUSIM/1 usando yacc. También hay otras herramientas basadas en Java para generar analizadores sintácticos ascendentes como ANTLR [ANTLi].

12.3 Análisis semántico

El **analizador semántico** detecta la validez semántica de las sentencias aceptadas por el analizador sintáctico. El analizador semántico suele trabajar simultáneamente al analizador sintáctico y en estrecha cooperación. Se entiende por **semántica** como el conjunto de reglas que especifican el significado de cualquier sentencia sintácticamente correcta y escrita en un determinado lenguaje [CUEV95b].

Las rutinas semánticas deben realizar la evaluación de los atributos de las gramáticas (véase apartado 11.5) siguiendo las reglas semánticas asociadas a cada producción de la gramática (véase ejemplo Ejemplo 11.5.1).

Por ejemplo para una expresión como:

(A + B) * (C + D)

el analizador semántico debe determinar que acciones pueden realizar los operadores aritméticos (+,*) sobre las variables A,B,C y D. Así cuando el analizador sintáctico reconoce un operador, tal como " + " o " * ", llama a una **rutina semántica** que especifica la acción que puede llevar a cabo. Esta rutina puede comprobar que los dos operandos han sido declarados, y que tienen el mismo tipo. También puede comprobar si a los operandos se les ha asignado previamente algún valor.

A continuación se muestra un método de la clase *Semantico* (escrito en lenguaje C++) que realiza comprobaciones de tipo de operadores aritméticos y del operador de asignación. Los tipos de datos que comprueba son entero (I), real (F) y carácter (C), devuelve el tipo del resultado. En caso de error devuelve el tipo error (E).

```
char Semantico::comprueba_exp_opArit_exp(char t1, char operador, char t3)
// Comprueba operaciones aritméticas y asignación de tipos simples
{
  if (t1==t3) { return t1;   }

  if (
     ( (t1=='C') && ((t3=='I')||(t3=='F')) )
     ||
     ( ((t1=='I')||(t1=='F')) && (t3=='C') )
    )
     {
      errores.semantico(3); //Operación con CHAR no permitida
      return ('E');
     }

 if (operador=='=')
    {
     if ((t3=='F') && (t1=='I'))
     errores.semantico(4); //Error en asignación de tipos
     return ('E');
     }
 if (t1=='I' && t3=='F') { return('F'); }
 if (t1=='F' && t3=='I') { return('F'); }
}
```

12.4 Tratamiento de errores

Los errores encontrados en las distintas fases de análisis se envian a un módulo denominado manejo de errores. En el caso más sencillo puede ser un subprograma al que se le invoca enviándole el código de error, y que se encarga de escribir un mensaje con el error correspondiente, y el número de línea donde se ha producido, así como de cortar el proceso de traducción. Si se desea construir un tratamiento de errores más completo, por ejemplo detectando todos los errores del programa fuente, el módulo se complica dado que los analizadores deben proseguir su trabajo con falta de datos.

A continuación se muestra un fragmento de código de un método de la clase *Errores* escrito en C++, para el tratamiento de errores sintácticos. En el caso que se presenta cada vez que se encuentra un error el compilador se detiene, y finaliza el proceso.

```
void Errores::sintactico(int codigo)
 {
      cout<<"LINEA "<<lexico.lineaActual();
      cout<<" ERROR SINTACTICO "<<codigo;
      switch (codigo)
       {
        case 1 :cout<<" :ESPERABA UN ;"<<endl;break;
        case 2 :cout<<" :ESPERABA UNA }"<<endl;break;
        case 3 :cout<<" :ESPERABA UN ="<<endl;break;
        case 4 :cout<<" :ESPERABA UN )"<<endl;break;
        case 5 :cout<<" :ESPERABA UN IDENTIFICADOR"<<endl;break;
        case 6 :cout<<" :INSTRUCCION DESCONOCIDA"<<endl;break;
        case 7 :cout<<" :ESPERABA UNA CONSTANTE"<<endl;break;
        case 8 :cout<<" :ESPERABA UNA M DE MAIN"<<endl;break;
        case 9 :cout<<" :ESPERABA UNA {"<<endl;break;
       ...
        default:
        cout<<" :NO DOCUMENTADO"<<endl;
       }
      exit(-(codigo+100));
 }
```

12.5 Tabla de símbolos

La tabla de símbolos es una estructura de datos que contiene toda la información relativa a cada identificador que aparece en el programa fuente. Los identificadores pueden ser de variables, tipos de datos, funciones, procedimientos, etc... Evidentemente cada lenguaje de programación tiene unas características propias que se reflejan en la tabla de símbolos [CUEV95c].

Cada elemento de la estructura de datos que compone la tabla de símbolos está formado al menos por el **identificador** y sus **atributos**. Los atributos son la información relativa de cada identificador. Los atributos habituales son:

- *Especificación del identificador*: variable, tipo de datos, función, procedimiento, etc...
- *Tipo*: en el caso de variables será el identificador de tipo (real, entero, carácter, cadena de caracteres, o un tipo definido previamente). En el caso de las funciones puede ser el tipo devuelto por la función. Los procedimientos se pueden marcar como funciones que no devuelven nada (*void*).
- *Dimensión o tamaño*. Puede utilizarse el tamaño total que ocupa una variable en bytes, o las dimensiones. Así en el caso de variables simples se coloca la dimensión cero, para los arrays la dimensión del array, en el caso de estructuras (*struct*) o registros (*record*) el número de campos o miembros, en el caso de funciones o procedimientos se puede almacenar el número de parámetros, etc...
- *Dirección de comienzo para generar el código objeto*. En el código máquina no hay nombre de identificadores, tan sólo direcciones de memoria para datos (*static*, *stack*, y *heap*) o para código (usadas por los procedimientos y funciones).
- *Listas de información adicional*. Contienen información de tamaño variable, por ejemplo: los tipos de los campos o miembros de las estructuras o registros, los tipos de los parámetros de estructuras o uniones.

Los atributos pueden variar de unos lenguajes a otros, debido a las características propias de cada lenguaje y a la metodogía de desarrollo del traductor.

La tabla de símbolos se crea por cooperación del análisis léxico y el análisis sintáctico. El análisis léxico proporciona la lista de los identificadores, y el análisis sintáctico permite rellenar los atributos correspondientes a cada identificador. El analizador semántico también puede rellenar algún atributo.

El analizador semántico y el generador de código obtienen de la tabla de símbolos la información necesaria para realizar su tarea.

Las operaciones fundamentales que se realizan en la tabla de símbolos son la **inserción** y **búsqueda** de información. En algunos lenguajes de programación también se realizan las operaciones **set** y **reset** que activan y desactivan los identificadores locales respectivamente. Dada la cantidad de veces que se accede a la tabla de símbolos es necesario que la estructura de datos que alberga la información, y los algoritmos de acceso sean optimizados al máximo. En general se utilizan métodos hash para el manejo de las tablas de símbolos.

Las tablas de símbolos constituyen el recipiente donde se almacena toda la información relativa al programa fuente en tiempo de compilación y por tanto se destruyen una vez finalizada la traducción. Tan sólo hay una excepción en el caso de que se activen opciones de depuración (*debugging*), en ese caso los compiladores graban en fichero la tabla de símbolos para poder dar información de los identificadores usados en el programa fuente en tiempo de ejecución. Los fabricantes de compiladores incluyen para esos casos información de la tabla de símbolos que emplean [BORL91].

12.6 Gestión de memoria en tiempo de ejecución

Otro aspecto importante que debe de realizar el compilador es la **gestión de memoria en tiempo de ejecución**. Los lenguajes de programación de tercera generación reparten la memoria en tres partes diferentes. En primer lugar está la **asignación estática de memoria** que se suele realizar para variables estáticas globales, por ejemplo un ARRAY declarado como variable global en un programa en Pascal, y que se caracteriza por que *se asigna en tiempo de compilación*, y las posiciones de memoria que ocupan se mantienen durante la ejecución. Los lenguajes COBOL y FORTRAN77 sólo permigen la gestión estática de memoria [CUEV95d].

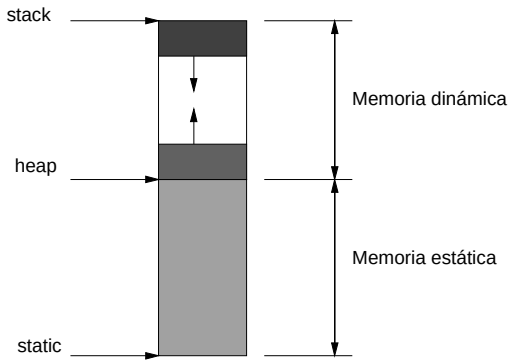


Fig. 30: Organización de la memoria en tiempo de ejecución

Otra forma de asignación de memoria es la llamada **asignación de memoria dinámica de pila o stack**, que se utiliza con las variables locales que sólo necesitan ocupar posiciones de memoria en el momento que se activan, por ejemplo el comienzo de la ejecución de un procedimiento o función en lenguaje Pascal se ocupan las posiciones de memoria necesarias para las variables locales, a la salida se vuelven a dejar libres. Evidentemente esta asignación se realiza en tiempo de ejecución, y la memoria reservada para este tipo de asignación se maneja con una estructura de pila LIFO. Este tipo de gestión de memoria es clave para los lenguajes recursivos, dado que es en esta pila donde se guardan las copias de los parámetros de cada llamada recursiva.

La última asignación de memoria es la llamada **asignación dinámica heap o de montón**, que se utiliza con las estructuras dinámicas de datos, dado que éstas se construyen en tiempo de ejecución, pudiendose asignar y liberar memoria en tiempo de ejecución, tal y como se realiza con los procedimientos estándar *New* y *Dispose* del lenguaje Pascal, o con las funciones *malloc()* y *free()* del lenguaje C. Habitualmente en muchas implementaciones la memoria *heap* crece en sentido inverso a la memoria *stack* (véase fig. 30).

12.7 Generación de código intermedio

La tarea de síntesis suele comenzar generando un **código intermedio**. El código intermedio no es el lenguaje de programación de ninguna máquina real, sino que corresponde a una **máquina abstracta**, que se debe de definir lo más general posible, de forma que sea posible traducir este código intermedio a cualquier máquina real. El objetivo del código intermedio es reducir el número de programas necesarios para construir traductores, y permitir más facilmente la **transportabilidad** de unas máquinas a otras. Supóngase que se tienen n lenguajes, y se desea construir traductores entre ellos. Sería necesario construir *n*(n-1)* traductores (fig. 31). Sin embargo si se construye un lenguaje intermedio, tan sólo son necesarios *2*n* traductores (fig. 32). Así por ejemplo un fabricante de compiladores puede construir un compilador para diferentes máquinas objeto con tán sólo cambiar las dos últimas fases de la tarea de síntesis (fig. 33).

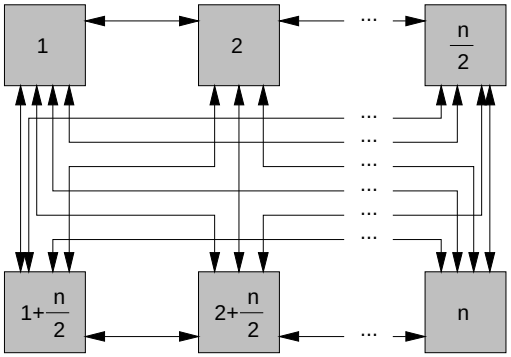


Fig. 31: n*(n-1) traductores entre n lenguajes

Las máquinas abstractas deben definirse completamente: por una parte se definirá su arquitectura y por otra su repertorio de instrucciones. La arquitectura de la máquina abstracta se elegirá de forma que contribuya a facilitar la portabilidad dentro del grupo de arquitecturas hacia las que previsiblemente se dirigirá el código objeto. Habitualmente las arquitecturas típicas de máquinas abstractas son: máquinas basadas en pila, basadas en registros, combinación de pilas y registros, orientadas a objetos. También se pueden clasificar desde el punto de vista de la cantidad y complejidad de sus instrucciones en máquinas CISC (*Complex Instruction Set Computer*) y RISC (*Reduced Instruction Set Computer*).

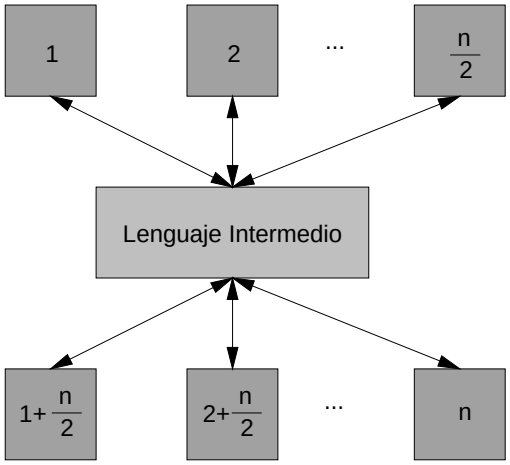


Fig. 32: 2*n traductores entre n lenguajes

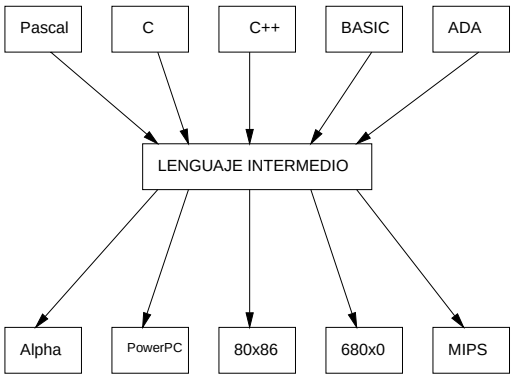


Fig. 33: Ejemplos de compiladores de distintos lenguajes para distintas máquinas

La forma de las instrucciones del código intermedio puede variar según sea el diseño de la máquina abstracta. Por ejemplo la expresión:

(A + B) * (C + D)

puede traducirse al siguiente conjunto de cuartetos:

(+ , A , B , T1)
(+ , C , D , T2)
(* , T1 , T2 , T3)

donde (+ , A , B , T1) se interpreta como suma A y B y coloca el resultado temporalmente en T1. El resto de las cuartetos se interpretan de forma similar. T1, T2, y T3 pueden ser registros de la máquina o posiciones de memoria temporales.

Otra forma de notación es la llamada **notación polaca inversa** (*Reverse Polish Notation* o RPN). Así la expresión anterior (A+B)*(C+D) en notación polaca inversa es:

A B + C D + *

De esta forma los operadores aritmeticos se colocan en el orden en que serán ejecutados, y además no es necesario el uso de paréntesis. Habitualmente la notación polaca va ligada a máquinas que utilizan una pila, y su representación interna es de la forma:

```
PUSH A
PUSH B
ADD
PUSH C
PUSH D
ADD
MUL
```

El código-P (abreviatura de código empaquetado, *packed*) utiliza operadores postfijos para el manejo de máquinas abstractas de pila. Los compiladores comerciales de Microsoft utilizan este tipo de código intermedio, su descripción puede consultarse en el manual *Programming Techniques*, capítulo 3 del compilador de C/C++ Versión 7.0 [MICR91, DIAZ92].

Otros traductores utilizan como código intermedio un lenguaje de medio nivel como puede ser el lenguaje C, que se caracteriza por su transportabilidad, por ejemplo el lenguaje Eiffel [MEYE88, MEYE92]. También se está utilizando por su eficiencia el lenguaje C++ para el desarrollo de lenguajes y herramientas orientadas a objetos [LEIV93].

La idea de un lenguaje intermedio universal no es nueva, y ya fue propuesta por T.B. Steel en 1961 [STEE61] con su célebre UNCOL (*UNiversal Communication Oriented Language*). El proyecto siempre fue bien recibido, pero la gran variedad de arquitecturas de ordenadores y otros intereses comerciales hicieron que el proyecto fracasase. Sin embargo la idea todavia permanece en la mente de muchas personas, y se puede decir que en muchas areas de la Informática los lenguajes C y C++ se han convertido en una especie de lenguaje intermedio universal. Actualmente la idea del lenguaje intermedio universal parece renacer debido a la necesidad de ejecutar aplicaciones independientes de plataforma en la red Internet. Así el código binario *bytecode* (ficheros con la extensión *.class*) de la máquina abstracta JVM (Java Virtual Machine) se está convirtiendo en el código intermedio universal, ya existen intérpretes de JVM para todos los sistemas operativos.

12.8 Generación de código

Una vez que se ha obtenido el código intermedio se pasa ensamblador o a código máquina, de una máquina real, en el caso de un compilador, o a otro lenguaje de programación en el caso de un traductor.

Por ejemplo la traducción de las tres cuadruplas:

(+ , A , B , T1)
(+ , C , D , T2)
(* , T1 , T2 , T3)

a un lenguaje ensamblador es:

```
LDA A         ;Carga el contenido de A en el acumulador
ADD B         ;Suma el contenido de B a lo que hay en el acumulador
STO T1        ;Almacena el contenido del acumulador en un registro temporal T1.
LDA C         ;Carga el contenido de C en el acumulador
ADD D         ;Suma el contenido de D a lo que hay en el acumulador
STO T2        ;Almacena el contenido del acumulador en un registro temporal T2
LDA T1        ;Carga el contenido de T1 al acumulador
MUL T2        ;Multiplica el contenido de T2 por lo que hay en el acumulador
STO T3        ;Almacena el contenido del acumulador en el registro temporal T3.
```

En el caso del uso de un lenguaje intermedio de pila:

```
PUSH A
PUSH B
ADD
PUSH C
PUSH D
ADD
MUL
```

Una traducción al lenguaje ensamblador de los microprocesadores 80x86 sería la siguiente. Para más detalle ver anexos XVI.7 y XVI.8.

```
;-------------- Código generado por PUSH A
XOR SI,SI ; Pone el registro índice SI a cero con un XOR
LEA SI, A ; Carga la dirección de A en SI
PUSH SI   ; Empuja la dirección de A en la pila
;-------------- Código generado por PUSH B
XOR SI, SI
LEA SI,B
PUSH SI
;-------------- Código generado por ADD
XOR AX,AX
XOR DX,DX
POP AX       ; Saca B de la pila y lo almacena en el registro AX
POP DX       ; Saca A de la pila y lo almacena en el registro DX
ADD AX,DX    ; Suma de enteros
```

```
PUSH AX      ; Empuja el resultado de A+B en la pila
;-------------- Código generado por PUSH C
XOR SI, SI
LEA SI,C
PUSH SI
;-------------- Código generado por PUSH D
XOR SI, SI
LEA SI,D
PUSH SI
;-------------- Código generado por ADD
XOR AX,AX
XOR DX,DX
POP AX       ; Saca D de la pila y lo almacena en el registro AX
POP DX       ; Saca C de la pila y lo almacena en el registro DX
ADD AX,DX    ; Suma de enteros
PUSH AX      ; Empuja el resultado de C+D en la pila
;-------------- Código generado por MUL
XOR AX,AX
XOR BX,BX
POP AX   ; Saca C+D
POP BX   ; Saca A+B
IMUL BX  ; Multiplicación de enteros: BX por AX
PUSH AX  ; Deja el resultado de la operación en el tope de la pila
```

También puede generarse código apoyándonos en una biblioteca de funciones o clases desarrollada en un lenguaje de alto o medio nivel. Las funciones o métodos de dicha biblioteca pueden ser llamadas desde ensamblador o código máquina (fig. 34). De esta forma no se tiene que generar todo el código a bajo nivel, tan sólo lo estrictamente necesario. Este método es menos óptimo, pero en muchos casos viene impuesto por la necesidad de contar con prototipos o con primeras versiones en plazos muy cortos de tiempo. Estos módulos se pueden ir depurando, de forma que si algunas funciones de las bibliotecas de alto nivel no están muy optimizadas, en sucesivas versiones se pueden ir sustituyendo por otras desarrolladas a bajo nivel. Esta forma de utilizar bibliotecas construidas con compiladores de terceros, ya ha dado lugar a algun intento de limitar el uso de los compiladores. Así Borland® trató de limitar el uso de su compilador Borland C++ 4.0 para el desarrollo de compiladores y sistemas operativos, pero la lluvia de protestas que recibió hizo que desistiese de tales propositos [STEV94].

La generación de código también puede hacerse directamente a código máquina binario, según las especificaciones del fabricante del microprocesador. Así Intel® ha definido la *Object module specification* para los módulos objeto (.OBJ) de la familia 80x86, véase el anexo I para una referencia más amplia.

En el código generado los fabricantes también ponen sus marcas en binario por si hubiera algún litigio sobre el uso correcto o incorrecto de sus compiladores. Sobre aspectos legales en desarrollo de software puede consultarse la obra de *Fishman* [FISH94].

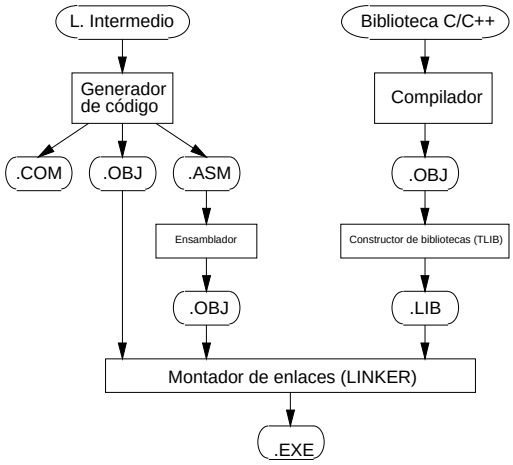


Fig. 34: Generación de código utilizando bibliotecas desarrolladas con lenguajes de alto nivel

12.9 Optimización de código

La finalidad de la optimización de código es producir un código objeto lo más eficiente posible. En algunos casos también se realiza una optimización del código intermedio. Algunas optimizaciones que se pueden realizar son la evaluación de expresiones constantes, el uso de ciertas propiedades de los operadores, tales como la asociativa, conmutativa, y distributiva; así como la reducción de expresiones comunes. En el ejemplo anterior, se puede optimizar el código por el uso de la propiedad conmutativa de la multiplicación:

```
LDA A
ADD B
STO T1
LDA C
ADD D
MUL T1
STO T2
```

Nótese que la expresión evaluada por el código optimizado es (C+D)*(A+B). La optimización anterior es independiente de la máquina, existen sin embargo optimizaciones que dependen de la arquitectura del ordenador, por ejemplo la asignación óptima de registros del microprocesador, o el uso de operaciones entre registros en vez de usar memoria y registros.

Habitualmente los compiladores permiten introducir las optimizaciones de código como opciones de compilación. Las opciones habituales son *velocidad de ejecución* y *tamaño del ejecutable*. Ambas optimizaciones son opuestas, es decir si se quiere mejorar la velocidad suele implicar incrementar el tamaño del ejecutable.

Las optimizaciones se pueden plantear como:

a) *Optimizaciones locales*. Son las que se realizan secuencialmente dentro de un bloque de código. Son las más sencillas de realizar.

b) *Optimizaciones de bucles*. Se realizan para optimizar bucles.

c) *Optimizaciones del flujo de un progama.* Es la más compleja y habitualmente se realizan por medio de teoría de grafos [DEJE89].

Algunas de las optimizaciones locales independientes de la máquina objeto se presentan a continuación:

- *Reducción o precálculo de constantes*. Consiste en realizar las operaciones con constantes antes de generar código. Así (2+3)*5 podría generar en una pila:

```
PUSH 2
PUSH 3
ADD
PUSH 5
MUL
```

  sin embargo si se hace la operación con las constantes sólo quedaría PUSH 25.

- *Fusión de constantes*. Suele realizarse con las constantes de cadena. Así el uso de la misma constante dos o más veces tan sólo genera código para una constante, el resto son la referencia a la misma constante.
- *Reacondicionamiento de instrucciones*. Por ejemplo el uso de las propiedades conmutativa y distributiva de los operadores aritmeticos puede permitir la reducción del código objeto. También se cambia en algunos casos el orden de evaluación, para generar código optimizado.
- *Comprobación de rangos y de desbordamiento de la pila*. Una vez que los programas compilados han sido probados, pueden quitarse del compilador las opciones de comprobación de rangos (habituales en los compiladores de Pascal) y las opciones de comprobación de desbordamiento de la pila. Al quitar estas opciones se deja de generar el código correspondiente para generar dichas comprobaciones.
- *Uso de operadores de desplazamiento de bits en vez multiplicación y división*. Las operaciones como *x*a*, donde a es una constante y potencia de 2, se puede generar código con operaciones de desplazamiento. Lo mismo ocurre para operaciones de la forma *x/a*.
- *Eliminación de código muerto*. El código que no se ejecuta no debe generar código. Así los bucles vacios no generan código.

Para la realización de la optimización de código dependiente de la máquina es fundamental conocer la arquitectura interna de la máquina en concreto y el rendimiento de cada una de sus instrucciones. Así para las arquitecturas 80x86 y Pentium® pueden consultarse las obras de optimización de código [ABRA94, SCHM94].

12.10 Front-end y back-end

Las fases descritas en los apartados anteriores se pueden agrupar en lo que se denomina **front-end** (parte delantera o frontal) y **back-end** (parte trasera).

El **front-end** son las fases, o parte de las fases, que sólo dependen en un principio del lenguaje fuente a compilar y son prácticamente independientes de la máquina. Normalmente incluye el análisis léxico, el análisis sintáctico, la creación de las tablas de símbolos, el análisis semántico, y la generación del código intermedio. También está incluido en el *front-end* la optimización de código independiente de la máquina, los procesos de manejo de errores de estas fases y la gestión de las tablas de símbolos.

El **back-end** incluye aquellas partes del traductor que dependen de la maquina objeto, y generalmente son independientes del programa fuente. Aquí se encuentran la generación de código, la optimización de código dependiente de la máquina, y el manejo de errores de estas fases. El *back-end* solamente está presente en los compiladores y los ensambladores.

12.11 Fases de un traductor

El modelo de traductor que se acaba de mostrar separa al traductor en distintos módulos, dando una idea de secuencialidad, sin embargo en muchos casos estos módulos se ejecutan simultáneamente. En la exposición anterior se ha omitido como se produce la secuencia de ejecución entre los distintos módulos del traductor.

Tal como se explicó en los apartados precedentes hay interacciones entre el analizador léxico y el analizador sintáctico. La primera posibilidad es que el analizador léxico genere un *token* para que sea procesado por el analizador sintáctico. Una vez que el analizador sintáctico lo ha procesado pedirá al analizador léxico el siguiente *token*, o en el caso de que no lo necesite se lo devuelva, repitiéndose este proceso durante el análisis del lenguaje fuente. Otra posibilidad es que el analizador léxico genere todos los *tokens* del programa fuente de una pasada, y los almacene en memoria o en un disco para que sean procesados posteriormente por el analizador sintáctico. En este caso esta parte del traductor se dice que es de **paso separado** (*separate pass*). Algunos traductores realizan todo el proceso en **un sólo paso** (traductores de un paso), mientras que otros realizan hasta más de 30 pasos (por ejemplo los primeros compiladores de PL/I de IBM). Los factores que influyen en el número de pasos de un compilador son los siguientes:

1. Complejidad del lenguaje fuente.
2. Distancia entre el lenguaje fuente y el lenguaje objeto.
3. Memoria mínima necesaria para compilar.
4. Velocidad y tamaño del compilador.
5. Velocidad y tamaño del programa objeto.
6. Necesidades de depuración (*debugging*).
7. Técnicas de detección y reconocimiento de errores.
8. Número de personas y tiempo disponible para realizar el traductor.

Las complejidad del lenguaje fuente también puede implicar que el compilador necesite realizar varias pasadas al lenguaje fuente para realizar el análisis del programa fuente, es decir es necesario leer varias veces el lenguaje fuente para obtener toda la información necesaria para la fase de análisis. Así por ejemplo la complejidad del lenguaje Eiffel obliga a sus compiladores a dar varias pasadas a los ficheros fuente [LEIV93].

Los compiladores orientados a educación de estudiantes de lenguajes de programación son compiladores de un paso. La optimización de código que se realiza en ellos es muy pequeña o no se hace, ya que la mayoría de los programas se compilarán muchas veces hasta que se pongan a punto. Ejecutándose una vez y descargándose. Las fases de análisis y generación de código se realizan a la vez. La parte más importante de este tipo de compiladores es la detección y reconocimiento de errores. Por ejemplo los compiladores Turbo Pascal® y Turbo C® de Borland.

Los compiladores profesionales, en general no suelen de ser de un paso, sino que son de varios pasos. Se suele cuidar mucho la fase de optimización de código.

Normalmente los fabricantes de compiladores comerciales suelen tener varias gamas de compiladores según el usuario al que vayan destinados. Así las gamas de compiladores Quick® de Microsoft o Turbo® de Borland van destinadas a estudiantes o a profesionales que busquen la velocidad de compilación sobre la optimización de código. Para profesionales que deseen una mejor calidad de código objeto, perdiendo velocidad de compilación, estos mismos fabricantes ofrecen las gamas Professional® y Borland® respectivamente.

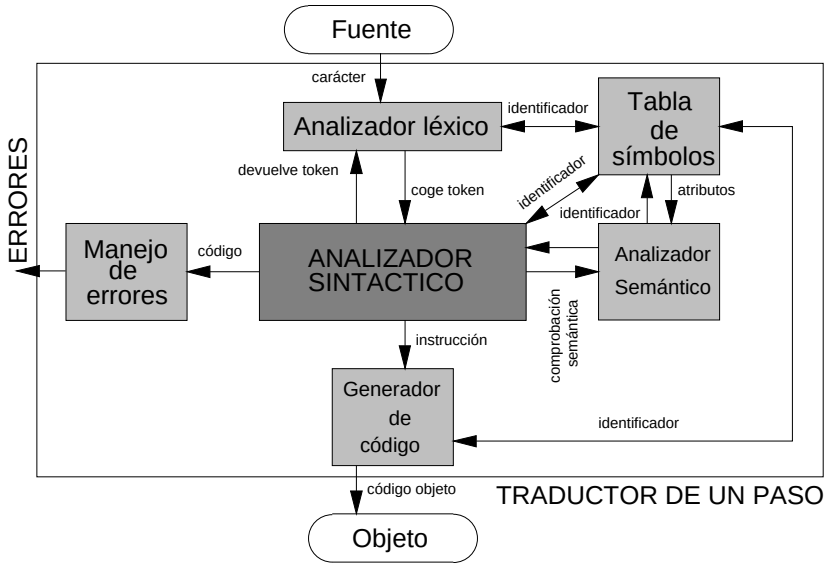


Fig. 35: Esquema general de un traductor de un paso

Los traductores de un paso y de una pasada única al lenguaje fuente son los más rápidos. Así los diseñadores de traductores eligen este esquema cuando la complejidad del lenguaje lo permite construirlos. En este caso el esquema habitual de los traductores está dirigido por el analizador sintactico, que constituye el centro del proceso de traducción o módulo principal, y el resto de los módulos cooperan a su alrededor (fig. 35).

12.12 Interfaz con el sistema operativo

La fase de montaje o enlace (*link*) permite construir un sólo programa objeto a partir de varios ficheros de código objeto. Estos ficheros son el resultado de varias compilaciones diferentes, o ficheros de una biblioteca de procedimientos o subrutinas disponibles por cualquier programa.

La fase de carga consiste en colocar físicamente las instrucciones de máquina y los datos en memoria, para ello deben de asignar espacio de memoria (*allocation*), y ajustar todas las posiciones dependientes de la dirección, para que se correspondan con el espacio asignado (*relocation*). Véase ANEXO XV El montador del sistema operativo MS-DOS®.

12.13 Validación

En la construcción de un procesador de lenguajes es fundamental el proceso de verificación de su funcionamiento, normalmente se realiza por medio de un conjunto de pruebas, que se tratan de sistematizar al máximo, comprobando cada uno de los módulos de que consta el traductor.

Por un lado están las pruebas de tipo α, que son las que se realizan dentro de la propia empresa de desarrollo, y por otra parte estan las pruebas de tipo β que se realizan por terceras partes: habitualmente especialistas, desarrolladores y usuarios en general.

Un criterio clásico de validación [SANC89] para las pruebas de tipo α es utilizar seis tipos de baterias de test, denominados cada una de ellas con una letra (*A, B, L, C, D,* y *E*). Las baterias de test son programas realizados en el lenguaje fuente y traducidos con el traductor desarrollado. Las baterias de programas deben diseñarse de la forma más general posible. *Un traductor que pase estas baterias de test no quiere decir que no tenga errores, tan solo que no se le han encontrado.*

12.13.1. Tests de tipo A

Son programas correctos que no deben dar errores en tiempo de compilación, y que el traductor debe tratar y generar a partir de ellos un código objeto. Un método habitual es hacer varios programas pequeños por cada tipo de instrucción, y cada cierto número de instrucciones un programa más completo que utilice todas ellas.

12.13.2. Tests de tipo B

Son programas con errores escritos a propósito para verificar si el traductor los detecta en tiempo de compilación. El traductor debe detectar cada error y enviar el mensaje correcto, indicando la línea donde se ha producido el error. Debe de construirse al menos un programa por cada tipo de error (léxico, sintáctico o semántico), aunque esto no es suficiente dado que el mismo tipo de error puede darse en lugares muy diferentes del traductor.

12.13.3. Test de tipo L

Son programas con errores escritos a propósito y que sólo se pueden detectar en la fase de montaje o enlace (*link*). Este tipo de test sólo se puede aplicar a los lenguajes que permiten la compilación separada de módulos. Los errores se detectarán cuando el *linker* trate de construir el programa ejecutable.

12.13.4. Test de tipo C

Pueden ser los mismos programas de test que los de tipo A, pero ahora se comprueban en tiempo de ejecución, y se debe verificar que el ejecutable realiza las instrucciones indicadas en el programa fuente. Normalmente se ampliarán los test de tipo A para comprobar aspectos específicos de la generación de código.

12.13.5. Test de tipo D

Este tipo de test comprueban las capacidades máximas de una implementación, por ejemplo número máximo de identificadores dentro de un bloque, agotamiento de la pila (*stack*) por recursividad, máximo nivel de anidamiento de subprogramas, etc... Estas capacidades máximas dependen de la implementación y del ordenador anfitrión.

12.13.6. Test de tipo E

Buscan posibles ambigüedades de instrucciones, es decir que un mismo tipo de instrucción pueda dar dos resultados diferentes. Si aparecen ambigüedades la implementación del traductor no es válida. Las ambigüedades deben ser eliminadas en la fase de diseño del lenguaje, pero dada la complejidad de los lenguajes puede darse el caso de que se escape alguna en la fase de diseño, en ese caso deben volverse a diseñarse las especificaciones léxicas, sintácticas y semánticas del lenguaje.

12.14 Entorno de un procesador de lenguaje

Los traductores y compiladores modernos suelen trabajar en entornos cómodos para el usuario, normalmente basados en ventanas y menus tanto de tipo texto como de tipo gráfico (*Graphic User Interface*, *GUI*). El entorno o interfaz de usuario se denomina en español latinoamericano como *medio ambiente*. Estos entornos suelen agrupar el editor, compilador, montadores, cargadores y depuradores facilitando al programador la tarea a desarrollar.

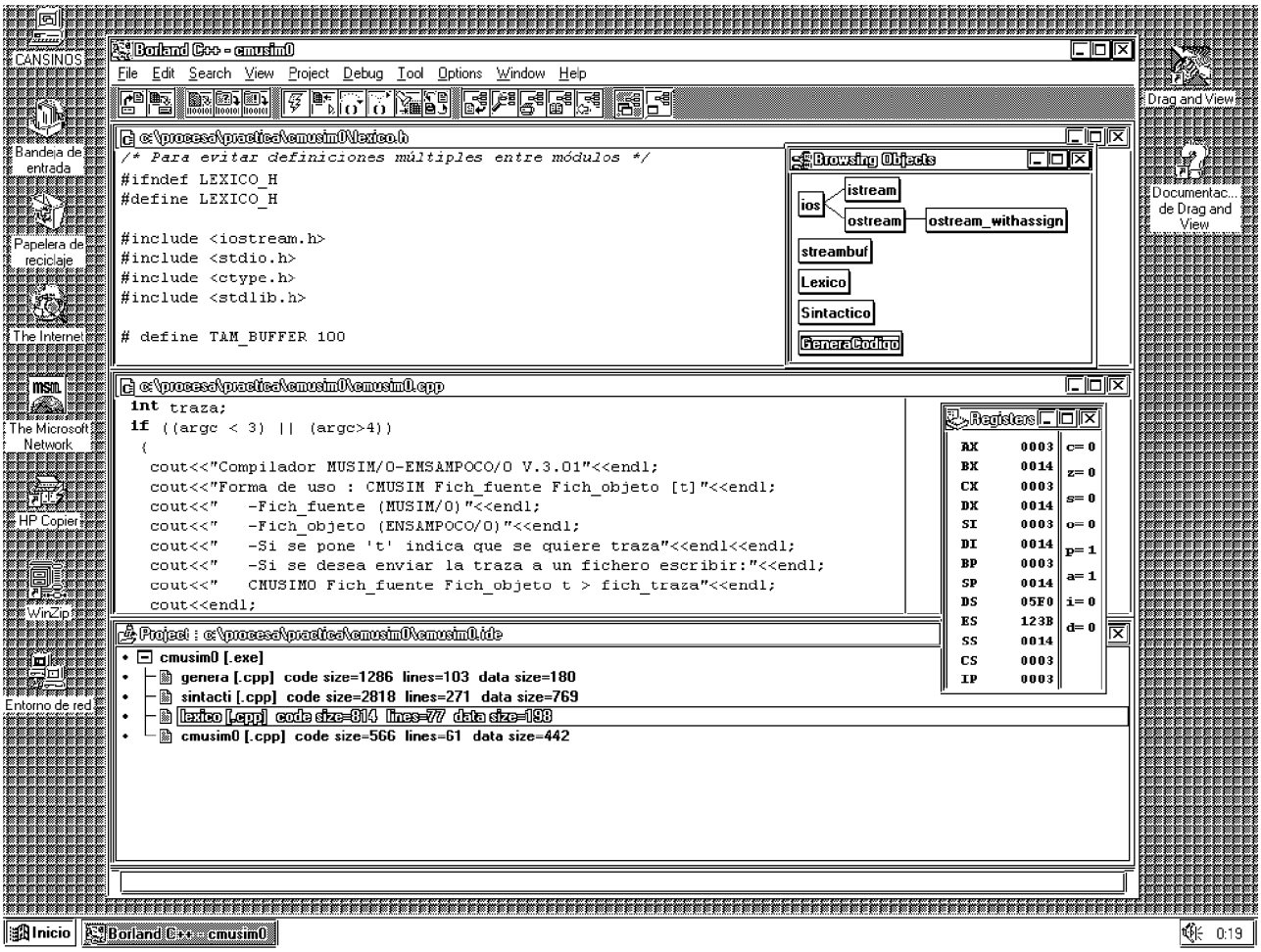


Fig. 36: Entorno del compilador Borland C++

12.15 Documentación

La documetación que debe acompañar a un traductor se puede agrupar en los siguientes manuales:

- **Manual del programador**. También se denomina *Guia del lenguaje* o *Manual de referencia del programador*. En algunos casos son varios tomos. Contiene la definición de todas las instrucciones del lenguaje, acompañadas de ejemplos, e indicando las restricciones del lenguaje o en el caso de estar normalizado sus diferencias. También se incluyen aquí la descripción de todas las funciones de las bibliotecas que acompañan al procesador de lenguaje.
- **Manual del usuario**. También se denomina *Guia del usuario*. Contiene la forma de instalar el compilador, la forma de uso, el entorno de trabajo, los errores que genera, como se depuran los errores, el manejo del editor (si lo tuviera) etc... También se incluyen aquí el la descripción del uso de depuradores, analizadores de rendimiento, constructores de bibliotecas, etc...
- **Manual de referencia técnica**: contiene la información relativa a como se construyó el compilador, con la información de cada uno de los módulos y la explicación de los códigos fuente. Este manual suele quedar disponible solamente a nivel interno para la empresa que ha desarrollado el compilador. Algunas partes que los desarrolladores consideren interesantes desde el punto de los usuarios pueden entregarse a estos, dando lugar a los denominados *Manuales de referencia técnica del usuario.*

13. INTÉRPRETES

Los intérpretes se definieron anteriormente, como programas que analizan y ejecutan simultáneamente el programa fuente, es decir no producen un código objeto, siendo su ejecución simultánea a la del programa fuente.

Se pueden clasificar desde el punto de vista de su estructura en varios tipos: intérpretes puros, intérpretes avanzados o normales, e intérpretes incrementales.

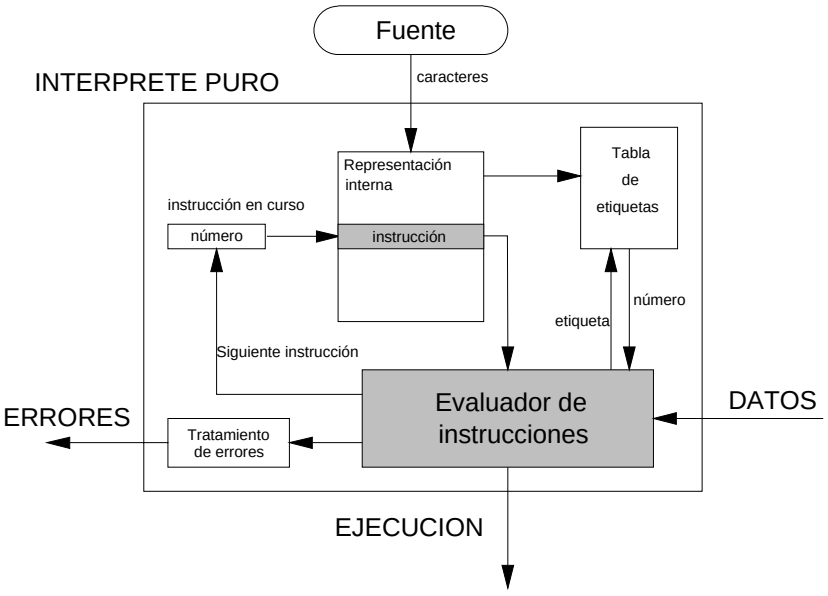


Fig. 37: Esquema general de un intérprete puro

13.1 Intérpretes puros

Los intérpretes puros *son los que analizan una sentencia y la ejecutan, y así sucesivamente todo el programa fuente*. Fueron los intérpretes desarrollados en la primera generación de ordenadores, pues permitían la ejecución de largos programas con ordenadores de memoria muy reducida, ya que sólo debián contener en memoria el intérprete y la sentencia a analizar y ejecutar. El principal problema de este tipo de intérpretes es que si a mitad del programa fuente se producen errores, se debe de volver a comenzar el proceso.

En la figura 37 se representa el esquema general de un intérprete puro, donde se puede observar que el lenguaje fuente se traduce a una *representación interna* (texto o binaria) que puede ser almacenada en memoria o en disco. Esta representación interna tiene todas las instrucciones numeradas o colocadas consecutivamente en estructuras de tamaño fijo (por ejemplo un *array* o posiciones consecutivas de memoria, o un fichero binario de estructuras de tamaño fijo). Mientras se realiza este paso se puede construir la *tabla de etiquetas*, que es una tablas que contiene una estructura donde están todas las etiquetas y su posición en el programa fuente (las etiquetas se utilizan tanto en las instrucciones de salto como en las llamadas a procedimientos y funciones). Una vez que este proceso ha finalizado, comienza la ejecución por la primera instrucción del código, que se envia al *evaluador de instrucciones*, éste la ejecuta (recibiendo datos si es necesario o enviendo un mensaje de error). El evaluador de instrucciones también determina la instrucción siguiente a ejecutar, en algunos casos previa consulta a la tabla de etiquetas. En el caso de que no haya saltos (*GOTO*) o llamadas a procedimientos o funciones se ejecuta la siguiente instrucción a la *instrucción en curso*.

El evaluador de instrucciones puede utilizar dos métodos de evaluación. El método clásico es la *evaluación voraz o ansiosa*, donde se evaluan las expresiones completamente. Otro método es la *evaluación perezosa*, evaluándose sólo la parte necesaria de la expresión (el resto no se evalúa).

Sobre la construcción de un intérprete puro de un mini-lenguaje orientado a objetos denominado *Bob* puede consultarse el artículo de *David Betz*, autor del intérprete XLISP [BETZ91].

13.2 Intérpretes avanzados

Los intérpretes avanzados o normales incorporan un *paso previo de análisis de todo el programa fuente*. Generando posteriormente un lenguaje intermedio que es ejecutado por ellos mismos. De esta forma en caso de errores sintácticos no pasan de la fase de análisis. En la figura 38 se representa el esquema general de un intérprete avanzado.

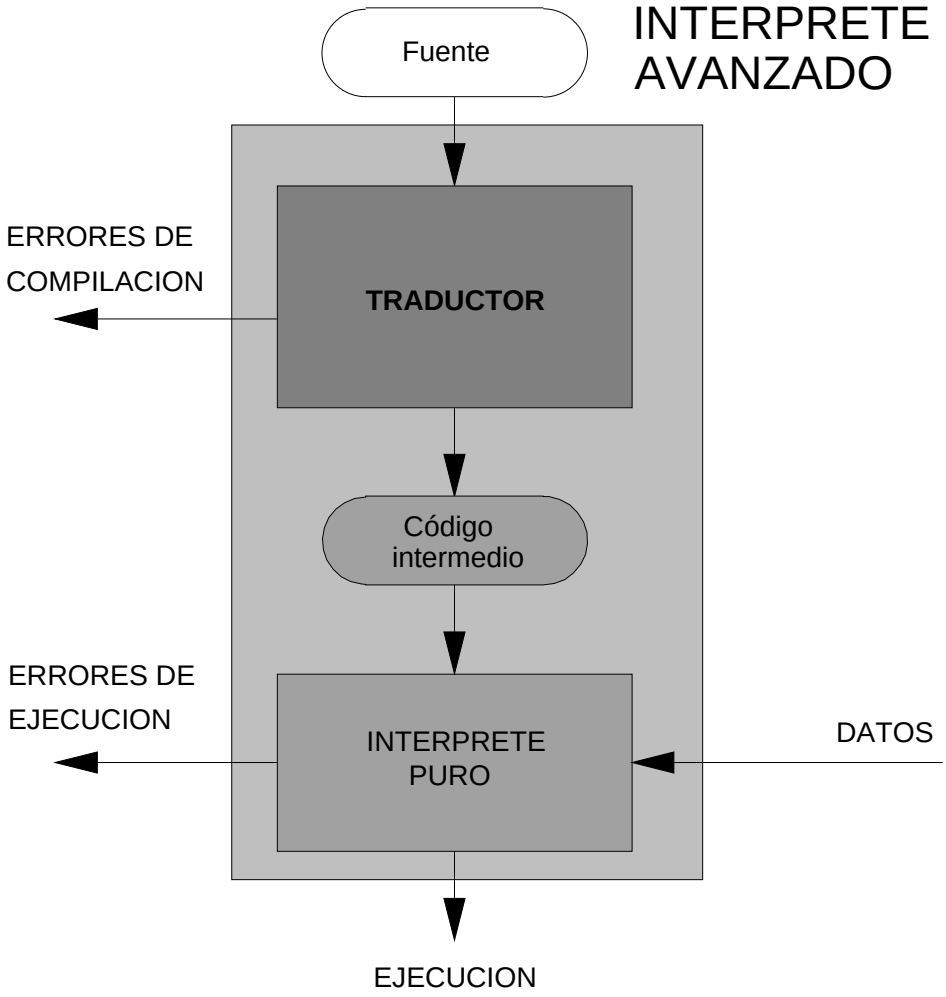


Fig. 38: Esquema general de un intérprete avanzado

Existen lenguajes que comenzaron siendo interpretados y que posteriormente se desarrollaron compiladores para ellos, por ejemplo Quick-BASIC®, pudiendo utilizarse el intérprete en la fase de elaboración y depuración de programas, y el compilador para la ejecución rutinaria del programa compilado

Otros ejemplos son los compiladores orientados a la enseñanza, como el TURBO PASCAL®, que generan un código intermedio que se ejecuta en memoria (es decir en este caso trabaja como intérprete avanzado), o también generan un programa ejecutable en disco (en este caso trabaja como compilador al crear un código objeto con la extensión *.exe*).

Un ejemplo de intérprete avanzado es el que utiliza el lenguaje Java [JAVAi]. Así un programa en lenguaje java (con la extensión *.java*) se compila y produce uno o varios ficheros con la extensión *.class*, estos ficheros están en un formato binario denominado *bytecode* [AZAÑ97] independiente de plataforma, que se interpreta posteriormente. Esto permite que el *bytecode* se ejecute en cualquier sistema operativo que disponga de un intérprete de *bytecode*. Dado que la mayor parte de los navegadores de Internet llevan inmerso un intérprete de *bytecode*, esto ha permitido al lenguaje Java ser uno de los más utilizados en aplicaciones que usen Internet.

Las aplicaciones de los intérpretes avanzados son:

- *Emulación de arquitecturas de ordenadores.*
- *Método para para facilitar la portabilidad de aplicaciones entre distintas arquitecturas de ordenadores.*
- *Fase de prueba de lenguajes y arquitecturas de máquinas abstractas, antes de construir los traductores a código objeto de una arquitectura determinada.*

13.3 Intérpretes incrementales

Algunos lenguajes no se pueden compilar, debido a que entre sus características pueden manejar objetos o funciones que no son conocidos en tiempo de compilación, ya que son creados en ejecución. Para este tipo de lenguajes existen los intérpretes incrementales, que permiten compilar los módulos completamente definidos, y recompilar en tiempo de ejcución los nuevos módulos. Están a medio camino entre los intérpretes puros y los intérpretes avanzados o normales.

Los intérpretes incrementales tienen gran interés en los lenguajes que permiten no definir los problemas completamente en tiempo de compilación. En estos casos se utilizan **evaluadores parciales** (*partial evaluators*) que toman como entrada el programa fuente junto con algunos datos (pero no todos), realizándose los cálculos que se pueden hacer con dicho subconjunto de datos, y produciendo una salida que contiene un *residuo* del programa fuente que se ha introducido [PAGA91].

Algunos fabricantes de software denominan a los intérpretes incrementales como compiladores incrementales, debido a que la palabra *intérprete* está desprestigiada, y prefieren acogerse a la parte del intérprete avanzado.

14. DISEÑO E IMPLEMENTACION DE LENGUAJES DE PROGRAMACION

El diseño y la implementación de lenguajes es una tarea compleja, así *Hoare* [SETH89] dice "*existen tantos criterios importantes pero conflictivos, que su reconciliación y solución adecuadas constituyen una gran tarea de ingeniería que demanda del diseñador del lenguaje un profundo conocimiento de todos los aspectos del arte de programar, la familiarización con una amplia gama de arquitecturas de ordenadores en las cuales pueda implantarse el lenguaje y el conocimiento de la gran diversidad de aplicaciones para las cuales éste pueda ser útil*". Los criterios que según *Hoare* debe complir un buen diseño de lenguaje son: "*sencillez, seguridad, rapidez de traducción, código (destino) eficiente y legibilidad*". Sobre la evolución y entresijos del desarrollo del lenguaje C++, puede consultarse la obra de *B. Stroustrup* [STRO94].

A continuación se enuncian las propiedades ideales de un buen lenguaje y una buena implementación [CAST93].

14.1 Propiedades de un buen lenguaje

- *Claridad, simplicidad y unidad de conceptos*
- *Léxico, sintaxis y semántica perfectamente definidas*
- *Consistencia con las notaciones habituales*
- *Facilidades para dar soporte a la abstracción y genericidad*
- *Independencia de la máquina y del sistema operativo*
- *Facilidades para dar soporte al control de errores y la verificación de programas*
- *Redundancias que ayudan a detectar errores*
- *Ortogonalidad que permita la mezcla de los elementos del lenguaje sin efectos laterales entre ellos.*

14.2 Características de una buena implementación

- *Normalización y portabilidad*
- *Entorno de programación*
- *Detección de errores*
- *Eficiencia del código generado (o eficiencia del intérprete)*
- *Mantenimiento y evolución de la implementación*
- *Documentación y soporte técnico*

Las características anteriores tanto del lenguaje como de la implementación se deben estudiar con detalle, cuando se elige un determinado procesador de lenguaje para un proyecto informático.

15. HERRAMIENTAS PARA LA CONSTRUCCION DE PROCESADORES DE LENGUAJE

La construcción de un procesador de lenguaje es una tarea compleja, y suelen usarse herramientas de desarrollo de software convencionales (control de la traza, puntos de ruptura, depuradores o *debuggers*, etc...). Sin embargo en el caso concreto de los procesadores de lenguaje se pueden añadir a estas herramientas otras más especializadas, que se han denominado con distintos nombres: compilador de compiladores (*compiler-compilers*), generadores de compiladores (*compilers-generators*), o sistemas de escritura de traductores (*translator-writing systems*). A continuación se muestra una lista de herramientas de este último tipo:

- **Generadores de analizadores léxicos** (*scanner generators*): Construyen automáticamente el código fuente para el análisis léxico a partir de una especificación de los tokens del lenguaje. Esta especificación está basada en el uso de expresiones regulares, mientras que la organización del generador es un autómata finito. La herramienta de este tipo más usada es el lex®, que viene incorporada con el sistema operativo UNIX®. Existen versiones comerciales para otros sistemas operativos, así por ejemplo para el sistema operativo MS-DOS, se puede encontrar PCLEX® de Abraxas® Software Inc. o como software de dominio público FLEX. En la Universidad de Oviedo se ha desarrollado GALEX [MART93].

- **Generadores de analizadores sintácticos** (*parser generators*): Construyen el código fuente del analizador a partir de la especificación de la gramática del lenguaje fuente. Las gramáticas deben cumplir ciertas condiciones. La herramienta de este tipo más usada es el *yacc*®, que viene incorporada con el sistema operativo UNIX. Existen versiones comerciales para otros sistemas operativos, así por ejemplo para el sistema operativo MS-DOS, se puede encontrar PCYACC (Abraxas Software Inc.) o como software de dominio público BISON. En la Universidad de Oviedo se ha desarrollado YACCOV [CABA91]. También se están desarrollando nuevas herramientas: JACK [JACKi], ANTLR [ANTLi], JavaCC, VisualParse++, LL(1) Tools, Yacc++, JavaCUP, etc.
- **Analizadores de gramáticas**: Dada una gramática especificada formalmente, verifican si es de un determinado tipo o no. Normalmente se utilizan para verificar las gramáticas LL(k) y LR(k). En la Universidad de Oviedo se ha desarrollado un analizador de gramáticas LL(1).
- **Máquinas de traducción dirigida por sintáxis** (*syntax-directed translation engines*): Producen un conjunto de rutinas que recorren el árbol sintáctico y generan código intermedio. Se basan en asociar una o más *traducciones* a cada nodo del árbol sintáctico.
- **Generadores automáticos de código** (*automatic code generators*): Estas herramientas trabajan con un conjunto de reglas que permiten la tradución del código en lenguaje intermedio al lenguaje objeto. Las reglas suelen ser reemplazar instrucciones de código intermedio por *patrones* que contienen las instrucciones equivalentes de la máquina objeto [MAUR90, LEWI90].
- **Analizadores de flujo** (*data-flow engines*): Estas herramientas suministran la información necesaria para realizar la optimización de código.

La reunión de todas estas herramientas constituyen los denominados compilador de compiladores, que a partir de unas especificaciones del lenguaje fuente a compilar y del lenguaje objeto, se genera automáticamente el código fuente del traductor.

16. APLICACIONES DE LOS PROCESADORES DE LENGUAJE

Las técnicas empleadas en la construcción de traductores, compiladores e intérpretes pueden aplicarse en la construcción de otras herramientas algunas de las cuales se presentan a continuación:

- **Editores sensibles al contexto**: Los editores permiten crear y modificar programas fuente, sin embargo los editores sensibles al contexto avisan al programador de posibles errores sintácticos cuando está escribiendo un programa fuente en un determinado lenguaje de programación. Actualmente la mayor parte de los compiladores incluyen un entorno de programación con un editor con *sintaxis resaltada por colores*.
- **Conversores de formatos**: Utilizan la tecnología de los traductores para convertir una descripción de ficheros en otra.
- **Preprocesadores**: Toman como entrada un conjunto de instrucciones y generan código en un lenguaje de alto o medio nivel.
- **Formateadores de código fuente**: Tienen como entrada un código fuente, y obtienen como salida el mismo código fuente mostrado de forma que se puede seguir perfectamente la estructura del programa.
- **Generadores de código**: Permiten desarrollar aplicaciones a partir de unas especificaciones muy compactas, que pueden ser tratadas como un lenguaje de aplicación [LEWI90].
- **Generadores de pantallas**: Son un caso particular de los generadores de código.
- **Verificación estática de programas**: Leen el programa fuente y lo analizan para descubrir errores potenciales, sin ejecutar dicho programa. Ejemplo *lint* (incorporado de forma estándar por UNIX) y *PC-lint*® (disponible comercialmente para DOS).
- **Formateadores de texto**: Reciben como entrada un texto con indicaciones de como se desea la salida y generan dicho texto formateado en un fichero, o para una determinada impresora. Pueden estar especializados en fórmulas matemáticas, químicas, escritura de música, etc... Ejemplos TROFF, EQN, etc...
- **Intérpretes de comandos de un sistema operativo**: Reciben las órdenes del sistema operativo, las analizan y las ejecutan. Ejemplo COMMAND.COM de MS-DOS.
- **Construcción de entornos operativos**: Es un caso particular del anterior en el cual las órdenes suelen recibirse en forma gráfica. Ejemplos WINDOWS, GEM, Macintosh, etc...
- **Intérpretes para consultar bases de datos**: Reciben las consultas de la base de datos, las analizan y las ejecutan. Ejemplos SQL, DBASE, etc...
- **Compiladores de silicio** (*silicon compilers*): Utilizan las mismas técnicas de construcción de traductores, compiladores e intérpretes pero implementadas en hardware [BROD92].
- **Procesamiento de lenguajes naturales**: Aplican las técnicas de construcción de traductores a los lenguajes naturales (Inglés, Castellano, etc...) permitiendo el análisis, comprensión y traducción.
- **Reconocimiento del habla**: Se realiza un análisis de los sonidos para construir las palabras.

- **Desarrollo de pequeños lenguajes**: Suelen ser pequeños lenguajes específicos para el problema que resuelve la aplicación informática dentro de la cual se desarrollan [FRAN91].

17. RESEÑA HISTORICA

Los primeros algoritmos conocidos fueron desarrollados en Mesopotamia (región de Irak) entre los años 3000 y 1500 a.C., cerca de la ciudad de Babilonia. Estos algoritmos eran secuencias de instrucciones, no tenían estructuras de control alternativas (escribían el algoritmo de diversas formas) ni repetitivas (escribian las instrucciones tantas veces como fuera necesario). Un estudio sobre estos algoritmos puede consultarse en el artículo *Ancient babylonian algorithms* [KNUT72].

El matemático griego *Euclides* en el siglo IV a.C. construyó un algoritmo para el cálculo del máximo común divisor de dos números naturales. Este algoritmo ya incluía estructuras de control iterativas.

Charles Babbage construyo entre los años 1820 y 1850 dos máquinas de computar, ninguna de las cuales fue terminada. De las dos, *la máquina analítica* fue la que más se parecía a los ordenadores digitales modernos. Los programas estaban escritos en lenguaje máquina y utilizaban ciertas tarjetas de operación y de variables (no había memoria). Junto a *Babbage* trabajó *Ada Augusta*, condesa de Lovelace, hija del poeta *Lord Byron*, que es reconocida por algunos como la primera programadora (en su honor se llamó Ada al lenguaje desarrollado bajo los auspicios del Ministerio de Defensa de los Estados Unidos).

En el fundamento lógico del reconocimiento y traducción de los lenguajes, siempre se debe de hacer referencia a la obra de *Alan Mathison Turing*. Su origen está en el problema de la decisión, planteado por *David Hilbert* en el Congreso Internacional de Matemáticos de 1900, en París. La publicación en la que *Turing* anunció su resultado ha tenido una significación y trascendencia que rebasó con mucho el problema inmediato al que se dirigía. Al atacar el problema de *Hilbert*, *Turing* se vio forzado a plantearse cómo dar al concepto de método una definición precisa. A partir de la idea intuitiva de que un método es un algoritmo, *Turing* hizo ver cómo esta idea puede refinarse y convertirse en un modelo detallado del proceso de computación, en el cual un algoritmo cualquiera es descompuesto en una secuencia de pasos simples. El modelo computacional resultante es una construcción lógica conocida por máquina de Turing, que puede reconocer todo tipo de lenguajes.

Otras aportaciones importantes utilizadas posteriormente en procesadores de lenguaje, se deben a las investigaciones de *Alonzo Church*, y dos de sus discípulos, *Stephen C. Kleene y J. Barkley Rosser*, sobre la lógica de la demostrabilidad, poniendo a punto un lenguaje formal coherente, bautizado por ellos como *lambda-cálculo*. Este lenguaje era útil para razonar sobre funciones matemáticas, tales como la raiz cuadrada, los logaritmos y cualesquiera otras funciones más complicadas que pudieran definirse (*Church* eligió la letra griega λ, equivalente a la "L" latina, para sugerir que su sistema formal es un lenguaje). El lenguaje de programación LISP, está inspirado en el *lambda-cálculo*.

Los primeras investigaciones relacionadas directamente con los procesadores de lenguaje, fueron el estudio del problema de lectura de fórmulas algebraicas complicadas y su interpretación. Así la notación polaca inversa (*Reverse Polish Notation*), o notación libre de paréntesis fue introducida por *J. Lukasiewicz* a finales de los años 20. Las fórmulas con paréntesis, en su caso más general fueron estudiadas por *Kleene* en 1934, y por *Church* en 1941. Continuando los estudios anteriores, *K. Zuse* especificó, en 1945, un algorítmo que determinaba cuando una expresión con paréntesis estaba bien formada. En 1951, *H. Rutishauser* describió el nivel de paréntesis de una fórmula algebraíca, como un perfil entre distintos niveles, que parte del nivel cero, y que va ascendiendo nivel a nivel, para descender por último, otra vez, a nivel cero. Todos estos estudios estaban situados dentro del campo de la Lógica.

La palabra inglesa *compiler* se atribuye a menudo a *Grace Murray Hopper* (1906-1992) [SAMM92], quien desarrolló la implementación de lenguajes de alto nivel como una recopilación de subrutinas de una biblioteca. En un principio se llamó compilador, a un programa que "*reunía*" subrutinas (compilaba). Así cuando en 1954 empezó a utilizarse el término "*compilador algebraico*", su significado es el actual.

En un principio, se facilitó el proceso de compilación restringiendo la notación, primeramente se suprimió la precedencia de los operadores, y por tanto se necesitaban muchos paréntesis. Otra forma era requiriendo el completo uso de paréntesis, pues se conocía una manera mecánica y simple de insertar los paréntesis, obteniéndose expresiones con el "máximo número de paréntesis", desarrollados por los estudios ligados al dominio de la Lógica. En 1952 *C. Böhm* centró su interés en este último método, y lo aplicó a las fórmulas secuencialmente. *Böhm* y *John Backus* estaban en el equipo de 18 personas que realizo el primer compilador de FORTRAN para IBM, en 1954. Este lenguaje se diseñaba al mismo tiempo que se desarrollaba.

Después del FORTRAN otros lenguajes de alto nivel empezaron a desarrollarse a finales de los años 50, y algunos de ellos todavía juegan un papel importante hoy en día, aunque han sido remodelados en revisiones periódicas. Este es el caso del LISP y COBOL que fueron diseñados originalmente en esa época.

Por otra parte, en 1957 *Kantorovic* mostró como se puede representar una fórmula con estructura de árbol, lo cual supuso el primer paso para la conexión con el análisis sintáctico de los lingüistas. La jerarquía de los lenguajes había sido introducida por *Chomsky* (1956), como parte de un estudio sobre lenguajes naturales. Su utilización para especificar la sintaxis de los lenguajes de programación la realizó *Backus*, con el borrador del que sería más tarde ALGOL 60.

Los compiladores diseñados en los años 50, compilaban secuencialmente fórmulas aritméticas. Todos usaban para almacenamiento intermedio de las partes procesadas una pila LIFO. En 1959 *Bauer y Samuelson* formalizaron esta técnica con los autómatas de pila.

Las técnicas fundamentales del análisis sintáctico se desarrollaron y formalizaron en los años 60. En un principio las investigaciones se centraron en el análisis sintáctico ascendente, aunque posteriormente se completaron con el estudio del análisis descendente.

El análisis sintáctico ascendente se basa en reconocer las sentencias desde las hojas hasta la raiz del árbol. *Floyd*, en 1963, fue el primero en introducir las gramáticas de precedencia, que dieron lugar a los compiladores basados en precedencia. En 1966 *McKeeman* obtuvo la precedencia débil como una generalización de la precedencia simple. Otro desarrollo en el análisis sintáctico descendente se debe a *Knuth*, que generalizó los autómatas de pila de *Bauer y Samuelson*, para reconocer los lenguajes generados por las gramáticas LR(k).

Los métodos descendentes de análisis sintáctico, se basan en construir el árbol sintáctico desde la raiz a las hojas, es decir en forma descendente. Los compiladores dirigidos por sintaxis, en la forma de análisis recursivo descendente fueron propuestos por primera vez de modo explícito por *Lucas* en 1961, para describir un compilador simplificado de ALGOL 60 mediante un conjunto de subrutinas recursivas, que correspondían a la sintásis BNF. El problema fundamental para el desarrollo de este método fue el retroceso, lo que hizo que su uso práctico fuera restringido. La elegancia y comodidad de la escritura de compiladores dirigidos por sintaxis fue pagada en tiempo de compilación por el usuario. Así mientras que con esta técnica tenían que ejecutarse 1000 instrucciones para producir una instrucción máquina, con las técnicas de los métodos ascendentes se tenían que ejecutar 50. La situación cambió cuando comenzó a realizarse análisis sintáctico descendente dirigido por sintaxis sin retroceso, por medio del uso de las gramáticas LL(1), obtenidas independientemente por *Foster* (1965) y *Knuth* (1967). Generalizadas posteriormente por *Lewis, Rosenkrantz y Stearns* en 1969, dando lugar a las gramáticas LL(k), que pueden analizar sintácticamente sin retroceso, en forma descendente, examinando en cada paso, todos los símbolos procesados anteriormente y los k símbolos más a la derecha. Evidentemente, tanto el análisis de las gramáticas LL(k) como el de las LR(k), es eficiente, y la discrepancia entre métodos de análisis sintáctico ascendente o descendente no va más allá de su importancia histórica.

La generación de código destino se efectuaba *ad hoc*, hasta comienzos de los sesenta, y no existían métodos formales para el desarrollo de esta tarea. Con la introducción de los lenguajes formales, la técnica general para realizar la generación de código, se basa en asociar a cada regla sintáctica unas reglas de traducción, llegándose al concepto actual de "traducción dirigida por sintaxis". También a principios de los sesenta se comenzaron a plantear técnicas de generación de código, mediante análisis sintáctico recursivo descendente. Así *Hoare* en 1962, describió un compilador de ALGOL 60, organizado como un conjunto de procedimientos, cada uno de los cuales es capaz de procesar una de las unidades sintácticas del informe de ALGOL 60.

A partir de mediados de los años sesenta se comienza la normalización por el American National Standards Institute (A.N.S.I.) de los lenguajes de programación, dándose un paso decisivo en la transportabilidad de los programas entre distintas máquinas y fabricantes.

Hasta finales de los años sesenta, la mayoría de los trabajos sobre teoría de lenguajes estaban relacionados con la definición sintáctica. La definición semántica es una cuestión mucho más compleja, por lo que recibió menor atención en la primera época de desarrollo. A finales de los años sesenta surgen las primeras definiciones semánticas elaboradas de lenguajes de programación, como la dada para PL/I por el laboratorio de IBM de Viena, en 1969.

La década de los setenta se caracteriza por la aparición de nuevos lenguajes de programación, resultado en parte de la formalización de las técnicas de construcción de compiladores de la década anterior, y por otra por el gran desarrollo del hardware. Entre los nuevos lenguajes se destaca el PASCAL, diseñado para la enseñanza de técnicas de programación estructurada, por *N. Wirth* en 1971. En el campo de la programación de sistemas destacan el lenguaje C, diseñado por *Kernighan y Ritchie* en 1973 [KERN78, KERN84, KERN88], y lenguaje Modula-2, diseñado por *Wirth* a finales de los setenta [WIRT88]. También hay que señalar la aparición de los pequeños sistemas (micro-ordenadores) y el abaratamiento de éstos.

La década de los ochenta comienza marcada por la aparición del lenguaje ADA, que pretende ser multipropósito, y con unas prestaciones muy elevadas. El diseño de este lenguaje se hizo según las directrices marcadas por el Ministerio de Defensa de los Estados Unidos. Es un lenguaje basado en el Pascal, que incluye características importantes tales como abstracción de datos, multitarea, manejo de excepciones, encapsulamiento y módulos genéricos. Para la construcción de sus compiladores se han utilizado las técnicas más avanzadas, y también se ha visto la necesidad de considerar nuevas técnicas de diseño de compiladores. Se avanza en el concepto de tipo abstracto de datos. Se desarrollan otras arquitecturas de computadoras, que faciliten el procesamiento paralelo. La tecnología VLSI, abre espectativas para el desarrollo de nuevas arquitecturas, y en el campo de compiladores de silicio. También hay que destacar en esta década el gran desarrollo de la Inteligencia Artificial, que ha conllevado el auge de los lenguajes LISP, PROLOG, Lazy ML (1984), MIRANDA (1985), Standard ML (1986) y Scheme así como el avance de investigaciones sobre procesamiento de lenguajes naturales.

La decada de los noventa se está iniciando con un fuerte impulso de los lenguajes orientados a objetos, de las interfaces gráficas de usuario (GUI), de la generación de código para las CPU paralelas, desarrollo para múltiples plataformas y de los lenguajes visuales [CHAN90, VARH94].

Sobre el desarrollo de compiladores y las probables tendencias en 1994 puede leerse el informe [HAYE94].

La aparición de Internet y la máquina abstracta JVM (Java Virtual Machine) puede influir notablemente en el desarrollo de compiladores al definirse un código intermedio común y portable entre distintintos sistemas operativos y ordenadores.

18. AMPLIACIONES Y NOTAS BIBLIOGRAFICAS

Como libro de cabecera para los conceptos teóricos de procesadores de lenguaje se recomienda la obra de *A. Aho et al.* [AHO86], aunque en muchos pasajes resulta dura y demasiado escueta. Sin embargo y sin lugar a dudas es el libro más completo. Este libro también denominado el libro del dragón, dado que tiene un dragón y un caballero con armadura en la portada. El dragón representa la complejidad de la tarea de construcción de compiladores. El caballero es el encargado de llevar a cabo la tarea de enfrentarse con el dragón, para lo cual se dota de una armadura (análisis), un escudo (traducción dirigida por sintaxis) y una espada (los generadores LALR).

Como complemento para la realización de prácticas se recomienda el libro de *A. I. Holub* [HOLU90], aunque todo el desarrollo de compiladores está basado en el lenguaje C, puede obtenerse código fuente es su página web [HOLUi].

También puede consultarse la obra de *Sánchez Dueñas y Valverde Andreu* [SANC89] en la parte de prácticas, en dicha obra se presenta el desarrollo de un compilador de un subconjunto de Pascal (*Gpasca*) utilizando el lenguaje Pascal para la implementación [SANC89].

En este texto se propone utilizar la *tecnología de orientación a objetos* para ayudar al caballero a vencer al dragón, y en especial el lenguaje C++ por su eficiencia en la tarea de desarrollo. Sobre el lenguaje C++ que va a ser la herramienta para construir los procesadores de lenguaje se recomiendan las obras de *B. Stroustrup* [STRO91, STRO94, STRO97] y la de *M. Ellis* y *B. Stroustrup* [ELLI90]. Si se desean aspectos más abstractos que la mera sintaxis se recomiendan las obras [DEVI93, KATR93, y KATR94]. Si se desean gran cantidad de ejemplos y ejercicios se puede consultar la obra [JOYA94]. Para una especificación completa de la sintaxis de un compilador de C++ comercial puede consultarse la versión castellana del compilador de Borland C++ Versión 4.02 [BORL94], aunque se aconseja para desarrollo una versión más moderna (Borland C++ 5.02 o C++ Builder).

Otras obra general de procesadores de lenguaje muy parecida a este texto es el libro de *Teufel et al.* [TEUF93].

19. EJERCICIOS RESUELTOS

Ejercicio 19.1: Compilador recursivo descendente de MUSIM/0

Con el objeto de clarificar algunos de los conceptos sobre diseño de compiladores, se desea construir un minicompilador de un lenguaje muy simple (*¡de alto nivel!*), que se denominará MUSIM (MUy_SIMple). La primera versión del lenguaje se denominará MUSIM/0. El lenguaje objeto será un lenguaje de bajo nivel denominado ENSAMPOCO (ENSAMbla POCO) de una máquina abstracta de pila, que en su primera versión se denominará PILINA/0 (PILA pequeñiNA).

19.1.1 Definición del compilador de MUSIM/0

El diagrama en T es el siguiente :

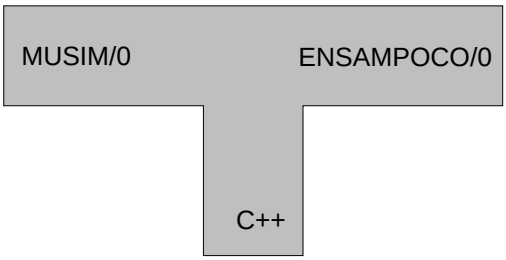


Fig. 39: Diagrama en T del compilador de $MUSIM/0_{C++}ENSAMPOCO/0$

19.1.2 Definición del lenguaje MUSIM/0

Se definirá el lenguaje indicando sus componentes léxicos, la gramática que describe su sintaxis, y las especificaciones semánticas.

a. Definición de los componentes léxicos

El lenguaje MUSIM/0 tiene los siguientes tipos de componentes léxicos o *tokens*:

- Identificadores, que sólo son nombres de variables y están compuestos por una única letra minúscula de rango a..z.
- Constantes numéricas de un sólo dígito, de rango 0..9.
- Operadores: +, -, *, / y %.
- Símbolo de asignación: = (igual).
- Paréntesis: (y).
- Separador de sentencias: ; (punto y coma).
- Indicadores de principio y fin de bloque: { y }.
- Palabras reservadas, están formadas por una letra mayúscula. Tan sólo son tres: R (lectura), W (escritura) y M (programa principal).

Puede observarse que en este lenguaje todos los *tokens* son de un sólo carácter.

b. Definición sintáctica

Se realizará formalmente mediante una gramática EBNF, que es LL(1), es decir el analizador sintáctico conoce que regla de producción ha de seguir con sólo conocer el token por el que comienza la sentencia.

La gramática en EBNF del lenguaje MUSIM/0 es:

```
<Programa>             ::= M "{" <bloque> "}"
<bloque>               ::= <sentencia> <otra_sentencia>
<otra_sentencia> ::= ; <sentencia> <otra_sentencia>
                     | <vacio>
<sentencia>            ::= <asignación> | <lectura> | <escritura>
<asignación>           ::= <variable> = <expresión>
<expresión>            ::= <término> <más terminos>
<más términos>         ::= + <término> <más términos>
                           | - <término> <más términos>
                           | <vacio>
<término>              ::= <factor> <más factores>
<más factores>         ::= * <factor> <más factores>
                           | / <factor> <más factores>
                           | % <factor> <más factores>
                           | <vacio>
<factor>               ::= ( <expresión> ) | <variable> | <constante>
<lectura>              ::= R <variable>
<escritura>            ::= W <variable>
<variable>             ::= a | b | c | ... | z
<constante>            ::= 0 | 1 | ... | 9
<vacío>                ::=
```

Puede observarse que este lenguaje sólo permite tres tipos de sentencias lectura[2], asignación y escritura. Tiene un sólo tipo de datos: entero. Las variables están formadas por una única letra minúscula, y las constantes son de un dígito. Tiene cinco operadores + (adición), - (diferencia), * (producto), / (división entera), y % (módulo). Se permite el uso de paréntesis. Atención, no se permite menos unario.

La precedencia de los operadores es la siguiente de más a menos precedencia:

1ª Evaluación de expresiones entre paréntesis ().

2ª Producto (*), división entera (/) y módulo (%).

3ª Adición (+) y diferencia (-).

Las reglas sintácticas siguientes en BNF:

```
<bloque>         ::= <sentencia> <otra_sentencia>
<otra_sentencia> ::= ; <sentencia> <otra_sentencia>
                     | <vacio>
```

también se podían escribir en una sola EBNF con:

```
<bloque> ::= <sentencia> { ; <sentencia> }
```

c. Definición semántica

MUSIM/0 sólo tiene un tipo de datos: los enteros.

Todos los operadores se aplican sobre enteros y los resultados son de tipo entero. Así el operador división / representa la división entera. Para obtener una división exacta se obtiene el cociente con el operador / y el resto con el operador módulo %.

2 En los lenguajes de programación habitualmente las instrucciones de lectura y escritura se definen como llamadas a procedimientos estándar (por ejemplo en el lenguaje Pascal).

d. Ejemplos de programas en MUSIM/0

A continuación se muestran ejemplos de programas en MUSIM/0:

```
1)
        M { R a; R b; c = a + b - 2 ; W c }
2)
        M {
        R a ; R b; R c;
        p = (a + b + c ) / 2 ;
        W p }
3)
        M
        {
        R a ;
        R b ;
        R c ;
        R d ;
        R e ;
        f = ( ( a + 5 ) * ( b - c ) ) % ( d / ( 8 - e ) ) ;
        W f
        }
4)
        M
        {
        R x; R y;
        d = x * x + y * y;
        i = x * x - y * y;
        c = (d + x) / (d -y)
        W d;
        W i;
        W c
        }
```

19.1.3 Definición del lenguaje objeto ENSAMPOCO/0

Como lenguaje objeto se utiliza un lenguaje intermedio que es un pequeño ensamblador, que se denomina ENSAMPOCO/0. Este ensamblador trabaja sobre una máquina de abstracta, que este caso particular es una máquina de pila. La máquina tendrá una memoria con 26 celdas cuyas direcciones se nombrarán con letras de la 'a' a la 'z', y una pila LIFO donde se realizarán las operaciones aritméticas.

La estructura de las instrucciones se define mediante la siguiente gramática EBNF, donde '\n' significa fin de línea:

```
<Programa>                          ::=.CODE '\n'
                                       {<Instrucción> '\n'}
                                       END
<Instrucción>                       ::= <Operador sin operando>
                                    | <Operador sobre dirección> <Dirección>
                                    | <Operador sobre constante> <Constante>
<Operador sin operando>             ::= NEG | STORE | LOAD | ADD | MUL | DIV | MOD
<Operador sobre dirección>          ::= PUSHA | INPUT | OUTPUT
<Operador sobre constante>          ::= PUSHC
<Dirección>                         ::= a | b | c | d | e | ··· | y | z
<Constante>                         ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A continuación se describe la forma de trabajo de las distintas instrucciones:

.CODE	Indica que comienza el código.
PUSHC	Coloca una constante en la pila. El operando es una cte.
PUSHA	Coloca en la pila la dirección de una variable. El operando es una variable, dado que las direcciones se denominan con el nombre de las variables.
LOAD	Asume que el último valor metido en la pila es una dirección. Esta dirección es extraida de la pila y en su lugar se pone el valor ubicado en dicha dirección. No tiene operando.
STORE	Usa los dos últimos elementos de la pila. Uno es la dirección de una celda de memoria y el otro el valor a almacenar en dicha celda. El último elemento de la pila es el valor y el otro la dirección. Después de ejecutada la instrucción, los dos elementos implicados son extraidos de la pila. No tiene operando.
NEG	Cambia el signo del último valor introducido en la pila, dejándolo en el mismo lugar. No tiene operando.
ADD	Opera con los dos últimos elementos introducidos la pila, extrayéndolos y dejando en su lugar el resultado. Por tanto la pila habrá disminuido en un elemento. No tiene operando.

MUL	Opera con los dos últimos elementos introducidos en la pila, extrayéndolos y dejando en su lugar el producto. Por tanto la pila habrá disminuido en un elemento. No tiene operando.
DIV	Opera con los dos últimos elementos introducidos en la pila, extrayendo primero el denominador y después el númerador y dejando en su lugar la división entera. Por tanto la pila habrá disminuido en un elemento. No tiene operando.
MOD	Opera con los dos últimos elementos introducidos en la pila, extrayendo primero el denominador y después el númerador y dejando en su lugar el módulo. Por tanto la pila habrá disminuido en un elemento. No tiene operando.
INPUT	Toma el valor del buffer de entrada, en este caso el teclado, y lo coloca en la dirección asignada a la variable. La pila no sufre cambios.
OUTPUT	Toma el valor de la dirección indicada y lo lleva al buffer de salida, en este caso la pantalla. La pila no sufre cambios.
END	Indica fin de programa.

A continuación se muestran ejemplos del lenguaje MUSIM/0, y su traducción al código intermedio ENSAMPOCO/0:

MUSIM/0
```
M {
R a ;
R b ;
z = a + b - 2 ;
W z }
```

ENSAMPOCO/0
```
.CODE
INPUT a
INPUT b
PUSHA z
PUSHA a
LOAD
PUSHA b
LOAD
ADD
PUSHC 2
NEG
ADD
STORE
OUTPUT z
END
```

En la figura 40 se representa gráficamente el proceso de ejecución de este programa en ENSAMPOCO/0.

El segundo ejemplo del lenguaje MUSIM/0:

MUSIM/0
```
M { R a ; R b ; R c ;
p = a + b + c ;
q = a *a + b * b + c * c;
r = ((p +q) * (p-q )) / ((p*p) - (q*q));
W p; W q ; W r }
```

El mismo programa en ENSAMPOCO/0 es el siguiente:

```
.CODE
INPUT a
INPUT b
INPUT c
PUSHA p
PUSHA a
LOAD
PUSHA b
LOAD
ADD
PUSHA c
LOAD
ADD
STORE
PUSHA q
PUSHA a
LOAD
PUSHA a
LOAD
MUL
PUSHA b
LOAD
PUSHA b
LOAD
MUL
ADD
PUSHA c
LOAD
PUSHA c
```

```
LOAD
MUL
ADD
STORE
PUSHA r
PUSHA p
LOAD
PUSHA q
LOAD
ADD
PUSHA p
LOAD
PUSHA q
LOAD
NEG
ADD
MUL
PUSHA p
LOAD
PUSHA p
LOAD
MUL
PUSHA q
LOAD
PUSHA q
LOAD
MUL
NEG
ADD
DIV
STORE
OUTPUT p
OUTPUT q
OUTPUT r
END
```

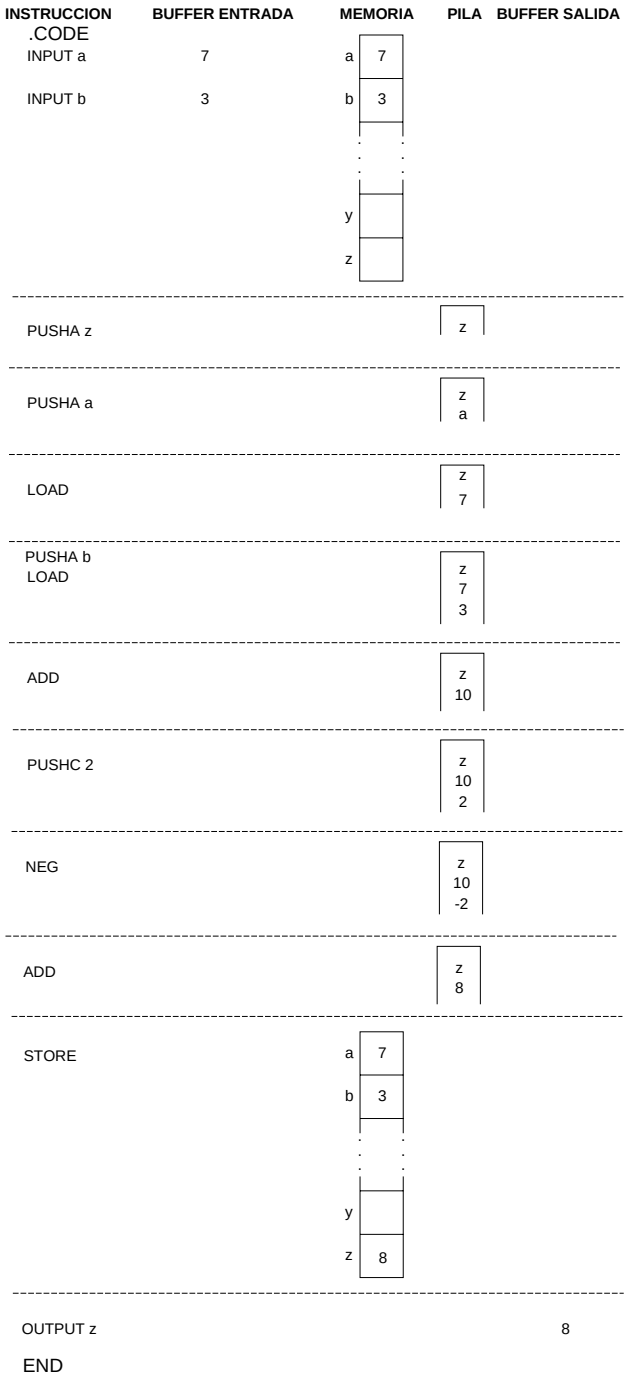


Fig. 40: Ejecución de un programa en ENSAMPOCO/0

19.1.4 Diseño y construcción del compilador

Se diseñara un compilador basado en la utilización de clases, como lenguaje se utilizará C++ debido a su eficiencia y portabilidad [HALL93, TSAI94, CONW94, HOLM95a, HOLM95b].

El compilador tendrá una opción de depuración, que permitirá la visualización de la traza del compilador, esta opción es muy util para comprender el compilador y para detectar fallos en su desarrollo.

Cada módulo del compilador será una clase. Durante el diseño preliminar sólo se diseñan las cabeceras *.h* de los distintos módulos. En la fase de implementación se desarrollan los cuerpos *.cpp* de cada módulo.

El compilador será de un paso, siendo el analizador sintáctico el módulo que dirige todos los procesos.

19.1.4.1 El analizador léxico

El análisis léxico debe de separar el fichero fuente en componentes léxicos o *tokens*, y enviarlos al analizador sintáctico. Habitualmente se envian los componentes léxicos y sus atributos. En este caso particular tan sólo se enviaran los tokens, ya que el atributo va implícito en el token (tan sólo se tiene el tipo de datos entero).

El analizador léxico también se encarga de abrir y cerrar el fichero fuente, y controlar si está produciendo la traza

A continuación se muestra la cabecera *lexico.h* del compilador.

```
/* Para evitar definiciones múltiples entre módulos */
#ifndef LEXICO_H
#define LEXICO_H

#include <iostream.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

# define TAM_BUFFER 100

class Lexico
 {
      char *nombreFichero;     // Nombre del fichero fuente de entrada
      FILE *entrada;           // Fichero de entrada
      int nl;                  // Número de línea
      int traza;               // Control de traza
      char buffer[TAM_BUFFER]; // Buffer auxiliar de caracteres
      int pBuffer;             // posicion en el buffer auxiliar

  public:
      Lexico (char *unNombreFichero, int una_traza=0);
      ~Lexico(void);
      char siguienteToken (void);
      void devuelveToken (char token);
      int lineaActual (void) {return nl;};
      int existeTraza(void) {if (traza) return 1; else return 0;}
  };
#endif
```

19.1.4.2 El analizador sintáctico

La técnica a utilizar es la denominada *recursivo descendente sin retroceso*, para su utilización se deben de cumplir dos requisitos:

- *El lenguaje fuente debe de estar definido con una gramática LL(1)*. Con un sólo *token* se decide la regla de la gramática que debe aplicarse. En el caso de que exista la posibilidad de una alternativa a <vacio> y el token no coincide se devuelve al analizador léxico.
- *El lenguaje de implementación debe de permitir la recursividad*.

Se define un método por cada símbolo no terminal de la gramática dentro de la clase *Sintactico*. Además se define un método para el tratamiento de errores sintácticos.

El analizador sintáctico estudiará el programa fuente, sentencia a sentencia, en primer lugar observará si la sentencia es de lectura o escritura (comienza por R o W), en caso contrario es una sentencia de asignación.

Detrás de cada sentencia de lectura o escritura esperará una variable y el final de sentencia o final de programa.

En las sentencias de asignación deberá encontrar en primer lugar una variable seguida del símbolo de asignación y una expresión. En este caso se debe de analizar la expresión según la gramática de MUSIM/0, tomando primero <término> y después <más términos>. Aplicándose recursivamente hasta el análisis completo de la expresión. La prioridad de los operadores ya está reflejada en la gramática. Puede observarse que no existe menos unario.

El analizador sintáctico debe detectar los errores sintácticos y enviar mensajes de error.

A continuación se muestra la cabecera *sintacti.h* del compilador, se ha elegido como decisión de diseño definir como miembros las clases *Lexico* y *GeneraCodigo*.

```
#ifndef SINTACTI_H
#define SINTACTI_H

#include "lexico.h"
#include "genera.h"

#include <stdlib.h>


class Sintactico
 {
  void programa (void);
  void bloque (void);
  void sentencia (void);
  void otra_sentencia (void);
  void asignacion(void);
  void lectura (void);
```

```
  void escritura(void);
  void variable(void);
  void expresion(void);
  void termino(void);
  void mas_terminos(void);
  void factor(void);
  void mas_factores(void);
  void constante(void);
  void errores (int codigo);
  Lexico lexico;
  GeneraCodigo generaCodigo;
  public:
       Sintactico(char *fuente, char *objeto, int traza);
       ~Sintactico(void);
  };
#endif
```

19.1.4.3 El analizador semántico

En este compilador no se realizará ya que todo es del mismo tipo (entero), no necesitándose comprobar otros aspectos del lenguaje.

19.1.4.4 La tabla de símbolos

No se utilizarán en este caso particular, pues todos los identificadores son variables y están definidas de antemano, son las letras minúsculas de a..z. El atributo tipo es entero en todas. La dirección de memoria viene designada con el nombre de la variable.

19.1.4.5 La generación de código intermedio

Se realiza línea a línea, simúltáneamente al análisis sintáctico. Las sentencias de lectura y escritura tienen traducción directa. Las sentencias de asignación deben de generar el código a medida que se hace el análisis. Toda le generación se basa en traducir las expresiones en notación algebraica, a notación postfija.

Se define una clase *GeneraCodigo* que contiene un método por cada instrucción de ENSAMPOCO/0, también se ocupa de abrir el fichero objeto y de cerrarlo.

```
#ifndef GENERA_H
#define GENERA_H

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>

class GeneraCodigo
 {
  char *nombreFichero; // Nombre del fichero objeto de salida
  FILE *salida;        // Fichero objeto de salida

  public:
       GeneraCodigo(char *unNombreFichero);
       ~GeneraCodigo();
       void code();
       void pushc(char constante);
       void pusha(char direccion);
       void load();
       void store();
       void neg();
       void add();
       void mul();
       void div();
       void mod();
       void input(char direccion);
       void output(char direccion);
       void end();
  };
#endif
```

19.1.4.6 La optimización de código intermedio

En este caso particular no se realiza.

19.1.5 Ejemplos y trazas

Los listados completos del compilador se encuentran en el anexo XVI. A continuación se muestra un ejemplo de programa de entrada, su traza por el compilador y el código objeto generado.

a. Un programa ejemplo: EJEM1.MUS

```
M
{
 R a;
 R b;
 R c;
 R d;
 z = ( a + b ) * ( c + d );
 W z
}
```

b. Traza de la compilación de EJEM1.MUS

A continuación se presenta la traza de compilación del programa EJEM1.MUS, en ella se va construyendo un árbol sintáctico similar al que se representó en la figura 29. Se propone al lector que dibuje el árbol sintáctico correspondiente a esta traza, en el mismo orden de la traza, para una mejor comprensión del análisis sintáctico descendente.

```
INICIO DE ANALISIS SINTACTICO
ANALISIS SINTACTICO: <PROGRAMA>
ANALIZADOR LEXICO: Lee el token M
.CODE
ANALIZADOR LEXICO: Lee el token {
ANALISIS SINTACTICO: <BLOQUE>
ANALISIS SINTACTICO: <SENTENCIA>
ANALIZADOR LEXICO: Lee el token R
ANALIZADOR LEXICO: Lee el token a
ANALISIS SINTACTICO: <LECTURA> a
INPUT a
ANALISIS SINTACTICO: <OTRA_SENTENCIA>
ANALIZADOR LEXICO: Lee el token ;
ANALISIS SINTACTICO: <SENTENCIA>
ANALIZADOR LEXICO: Lee el token R
ANALIZADOR LEXICO: Lee el token b
ANALISIS SINTACTICO: <LECTURA> b
INPUT b
ANALISIS SINTACTICO: <OTRA_SENTENCIA>
ANALIZADOR LEXICO: Lee el token ;
ANALISIS SINTACTICO: <SENTENCIA>
ANALIZADOR LEXICO: Lee el token R
ANALIZADOR LEXICO: Lee el token c
ANALISIS SINTACTICO: <LECTURA> c
INPUT c
ANALISIS SINTACTICO: <OTRA_SENTENCIA>
ANALIZADOR LEXICO: Lee el token ;
ANALISIS SINTACTICO: <SENTENCIA>
ANALIZADOR LEXICO: Lee el token R
ANALIZADOR LEXICO: Lee el token d
ANALISIS SINTACTICO: <LECTURA> d
INPUT d
ANALISIS SINTACTICO: <OTRA_SENTENCIA>
ANALIZADOR LEXICO: Lee el token ;
ANALISIS SINTACTICO: <SENTENCIA>
ANALIZADOR LEXICO: Lee el token z
ANALIZADOR LEXICO: Recibe en buffer el token z
ANALISIS SINTACTICO: <ASIGNACION>
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token z
PUSHA z
ANALIZADOR LEXICO: Lee el token =
ANALISIS SINTACTICO: <EXPRESION>
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token (
ANALISIS SINTACTICO: <EXPRESION>
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token a
ANALIZADOR LEXICO: Recibe en buffer el token a
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token a
PUSHA a
LOAD
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token +
ANALIZADOR LEXICO: Recibe en buffer el token +
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token +
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token b
ANALIZADOR LEXICO: Recibe en buffer el token b
```

```
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token b
PUSHA b
LOAD
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ADD
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ANALIZADOR LEXICO: Lee el token )
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token *
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token (
ANALISIS SINTACTICO: <EXPRESION>
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token c
ANALIZADOR LEXICO: Recibe en buffer el token c
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token c
PUSHA c
LOAD
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token +
ANALIZADOR LEXICO: Recibe en buffer el token +
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token +
ANALISIS SINTACTICO: <TERMINO>
ANALISIS SINTACTICO: <FACTOR>
ANALIZADOR LEXICO: Lee el token d
ANALIZADOR LEXICO: Recibe en buffer el token d
ANALISIS SINTACTICO: <VARIABLE>
ANALIZADOR LEXICO: Lee el token d
PUSHA d
LOAD
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ADD
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token )
ANALIZADOR LEXICO: Recibe en buffer el token )
ANALIZADOR LEXICO: Lee el token )
MUL
ANALISIS SINTACTICO: <MAS_FACTORES>
ANALIZADOR LEXICO: Lee el token ;
ANALIZADOR LEXICO: Recibe en buffer el token ;
ANALISIS SINTACTICO: <MAS_TERMINOS>
ANALIZADOR LEXICO: Lee el token ;
ANALIZADOR LEXICO: Recibe en buffer el token ;
STORE
ANALISIS SINTACTICO: <OTRA_SENTENCIA>
ANALIZADOR LEXICO: Lee el token ;
ANALISIS SINTACTICO: <SENTENCIA>
ANALIZADOR LEXICO: Lee el token W
ANALIZADOR LEXICO: Lee el token z
ANALISIS SINTACTICO: <ESCRITURA> z
OUTPUT z
ANALISIS SINTACTICO: <OTRA_SENTENCIA>
ANALIZADOR LEXICO: Lee el token }
ANALIZADOR LEXICO: Recibe en buffer el token }
ANALIZADOR LEXICO: Lee el token }
END
FIN DE ANALISIS SINTACTICO
FIN DE COMPILACION
```

c. El código generado: EJEM1.POC

```
.CODE
INPUT a
INPUT b
INPUT c
INPUT d
PUSHA z
PUSHA a
LOAD
PUSHA b
LOAD
ADD
```

```
PUSHA c
LOAD
PUSHA d
LOAD
ADD
MUL
STORE
OUTPUT z
END
```

Ejercicio 19.2: Traductor ENSAMPOCO/0 a ensamblador 80x86

Se desea construir un traductor de ENSAMPOCO/0 a ensamblador 80x86. Con este ejemplo se pretende ilustrar:

- La construcción de un traductor entre lenguajes del mismo nivel, realizándose una traducción intrucción a instrucción, y sin análisis sintáctico complejo.
- La generación de código para una máquina real a partir de un lenguaje intermedio de una máquina abstracta.

Solución: El programa toma el fichero fuente en ENSAMPOCO/0, y genera el fichero objeto en ensamblador 80x86. Se ha construido una clase denominada *Traductor*, que incorpora un método por cada instrucción de Ensampoco/0. También incorpora un método denominado *traduce* que es el cuerpo del traductor y se basa en una estructura multialternativa que decide la instrucción a traducir. Observese que en los lenguajes de bajo nivel es muy fácil la discriminación entre las instrucciones. El constructor de la clase se encarga de abrir los ficheros fuente y destino. El destructor se encarga de cerrarlos. Tambíen hay un método denominado *lexico* que se encarga de extrae las instrucciones del fichero fuente. El tratamiento de errores lo realiza el método denominado *errores*. A continuación se muestra el archivo cabecera *traducto.h*

```
#ifndef TRADUCTO_H
#define TRADUCTO_H

#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define TAM_MAX_OP 7

class Traductor
 {
  FILE *entrada, *salida;
  char operador[TAM_MAX_OP];
  char direccion;

  void lexico(void);      // Lee del fuente operador y direccion
  void traducir(void);    // Traduce cada tipo de instrucción
  void code(void);        // Un método por cada instrucción */
  void pushc(void);
  void pusha(void);
  void load(void);
  void store(void);
  void neg(void);
  void add(void);
  void mul(void);
  void div(void);
  void mod(void);
  void input(void);
  void output(void);
  void end(void);
  void errores(int);    //Tratamiento de errores

  public:
  Traductor(char *unNombreFicheroEntrada, char *unNombreFicheroSalida);
  ~Traductor(void);
};

#endif
```

La generación de código comienza con la instrucción .CODE que escribe en primer lugar en el fichero objeto la cabecera del programa Ensamblador, y posteriormente va leyendo línea a línea el fichero fuente y traduciéndo cada una de las instrucciones directamente al código equivalente. Por último en la instrucción END se escribe el epílogo con los procedimientos auxiliares. Se podría también optimizar este código comprobando que se incluyen tan solo los procedimientos ensamblador que se utilizan.

El programa objeto obtenido se puede pasar a código máquina (.OBJ) con un ensamblador comercial, por ejemplo MASM® de Microsoft® o TASM® de Borland®, y posteriormente enlazarse con un montador de enlaces comercial como LINK® de Microsoft® o TLINK® de Borland®, obteniéndose el ejecutable (.EXE) [GOOD92].

El código del programa traductor completo está en el anexo XVI.2. En el anexo XVI también se muestran ejemplos de salidas en ensamblador del traductor. Se puede escribir un programa en el lenguaje de mandatos del sistema operativo para que pase directamente de un programa en MUSIM/0 (*.MUS*) a un fichero ejecutable (*.EXE*), como ejemplo se muestra en el anexo XVI.4 el fichero *compila.bat*.

Ejercicio 19.3: Intérprete puro PIPO

Aquí se desarrolla un intérprete puro, para trabajar a modo de calculadora de expresiones en RPN, notación polaca inversa (*Reverse Polish Notation*). Cada expresión será una sentencia, y cada vez que el operador pulse retorno de carro <intro>, la expresión se evaluará. Se utilizará una pila LIFO, último en entrar primero en salir (*last in first out*), para realizar los cálculos y almacenar los valores. El único tipo de datos es el real. Al lenguaje interpretado se le denomina PIPO (PIla_POlaca).

19.3.1. Definición de los componentes léxicos

La implementación se basa en analizar la cadena de caracteres que constituye cada sentencia. Los *tokens* en este caso sólo pueden ser:

- constantes reales con punto decimal y sin exponente.
- constantes enteras
- uno o varios espacios en blanco separan las constantes
- \n puede separar constantes u operadores
- operadores de un carácter: +, -, *, y /.
- Evaluador de expresiones: =
- otros: c (borra la pila), h (ayuda), p (muestra la pila) y f (finaliza sesión).

19.3.2. Definición de la sintaxis

La sintaxis de las instrucciones es la siguiente:

```
<instrucción>          ::= <calcula> | <mandato>
<calcula>              ::= { <expresión> } = '\n'
<expresión>            ::= <constante> | <operador>
<constante>            ::= entera | real
<operador>             ::= + | - | * | /
<mandato>              ::= h | H | c | C | p | P | f | F
```

19.3.3. Definición semántica

La calculadora maneja dos tipos de datos enteros y reales. Si en las operaciones los dos operandos son enteros el resultado es real, pero si los operandos son uno de cada tipo: entero y real el resultado también es de tipo real.

19.3.4. Evaluador

El evaluador se rige según la normativa siguiente:

- Las constantes según se detectan son introducidas en la pila LIFO.
- Dado que se trabaja con una pila no hay precedencia de operadores. Los operadores trabajan sobre los dos últimos operandos introducidos en la pila sacándolos realizando la operación y metiendo el resultado en la pila.
- El operador evaluación =, muestra el contenido del tope de la pila sin modificarlo.
- Los mandatos se ejecutan directamente sin afectar para nada al contenido de la pila, excepto el mandato borrar pila.

19.3.5. Diseño y construcción del intérprete

La estructura del intérprete está basada en la clase *Rpn*, y se basa en trabajar con un *evaluador* de expresiones y mandatos que se encarga de ejecutarlos. Como elemento auxiliar se utiliza una pila LIFO, con las operaciones de apilar (*push*) y desapilar (*pop*). Hay una parte dedicada a la lectura de teclado usando un *buffer*. También se incluye un método de tratamiento de errores. El diseño preliminar se puede ver a continuación en la cabecera *rpn.h*, los programas completos están el anexo XVII.

```
#ifndef RPN_H
#define RPN_H

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>

#define MAX_INS 20        // Tamaño máximo de una instrucción
#define NUMERO '0'        // Atributo de la instrucción número
#define GRANDE '9'        // La instrucción es demasiado grande
#define TAM_BUFFER 100    // Tamaño del buffer del analizador léxico
```

```
class Rpn
{
 double *tope;              // Señala al tope de la pila
 double *inicio;            // Señala el principio de la pila
 double *fin;               // Señala el final de la pila
 int tamPila;               // Tamaño de la pila
 char instruccion[MAX_INS]; // Instrucción
 int atributo_ins;          // Atributo de la instrucción
 char buffer[TAM_BUFFER];   // Buffer del analizador de instrucciones
 int pBuffer;               // Posición en el buffer

 void evaluador(void);       // Evaluador de instrucciones
 void prompt(void);          // Prompt del intérprete
 void ayuda (void);          // Información de ayuda
 void errorFatal (int cod);  // Errores que implican finalización
 void avisoError (int cod);  // Errores que dan un mensaje y continuan
 int toma_instruccion(void); // Toma la siguiente instrucción
 int toma_char(void);        // Coge un carácter para formar la instrucción
 void devuelve_char(int c);  // Devuelve un carácter al buffer
 void borra(void);           // Borra el contenido de la pila
 double push(double);        // Empuja un valor en la pila
 double pop(void);           // Saca un valor de la pila

  public:
  Rpn(int tam_pila);
 ~Rpn(void);
};
#endif
```

Ejercicio 19.4: Intérprete de ENSAMPOCO/0

Construir un intérprete del lenguaje ENSAMPOCO/0, o enunciado de otra forma construir un emulador de la máquina abstracta PILINA/0 y de su repertorio de instrucciones ENSAMPOCO/0.

Solución: El intérprete del lenguaje ENSAMPOCO/0 se basa en simular su máquina abstracta PILINA/0 mediante un array (memoria) y una pila LIFO. Se diseña una clase denominada *Interprete*.

El array contiene 26 elementos, que corresponden a las 26 posiciones de memoria que ocupan las 26 posibles variables de la máquina abstracta. La pila LIFO será utilizada para realizar las distintas operaciones. Se diseña un *evaluador* de instrucciones, y por cada instrucción se escribe un método que la simula. También se incorpora un método *lexico* para tomar las instrucciones del fichero fuente, y un tratamiento de errores. A continuación se muestra el fichero cabecera *interpre.h*, el código completo está en el anexo XVI.3.

```
#ifndef INTERPRE_H
#define INTERPRE_H

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>

#define TAM_MAX_INS 7

struct pila {
           char valor;
           struct pila *sucesor;
           };

typedef struct pila ELEMENTO;
typedef ELEMENTO *PUNTERO;

class Interprete
 {
  FILE *entrada;                 //Descriptor del fichero de entrada
  char instruccion[TAM_MAX_INS];
  char parametro;
  PUNTERO tope;       //puntero al tope de la pila
  int memoria[26];    //memoria: array de 'a'..'z'de int

  void lexico(void);    // Coge las instrucciones del fichero fuente
  void evaluador(void); // Evalúa las expresiones
  void error(int);      // Manejo de errores

  void code(void);      // Instrucciones de ENSAMPOCO/0
  void pusha(void);
  void pushc(void);
  void load(void);
  void store(void);
  void neg(void);
  void add(void);
  void mul(void);
  void div(void);
  void mod(void);
```

```
  void input(void);
  void output(void);
  void end (void);
  public:
  Interprete (char *unNombreFicheroEntrada);
  ~Interprete (void);
 };
#endif
```

La combinación del intérprete de ENSAMPOCO/0 y el compilador de MUSIM/0 produce un intérprete avanzado de MUSIM/0, puede enlazarse todo por medio de unun programa de mandatos del sistema operativo, por ejemplo un programa *.BAT* en DOS, como *interpre.bat* que se muestra en el anexo XVI.5. También se puede escribir un programa completo que los ejecute consecutivamente.

Ejercicio 19.5: Definición del lenguaje MUSIM/1

El lenguaje MUSIM/1 es el lenguaje MUSIM/0 con las siguientes ampliaciones: operador menos unario (-) y operador potenciación (^).

Solución: En primer lugar tenemos un nuevo *token* ^ y otro *token con dos significados* en función del contexto. La inclusión de dos operadores nos obliga a redefinir la tabla de precedencia de operadores. Se elige la solución más parecida a la que incorpora el lenguaje matemático:

1ª ( )

2ª ^

3ª - unario

4ª * / %

5ª + -

Las reglas de precedencia se pueden reflejar en una gramática LL(1) de la siguiente forma:

```
<PROGRAMA> ::= M  "{"  <BLOQUE>  "}"
<BLOQUE> ::= <SENTENCIA> <OTRA_SENTENCIA>
<OTRA_SENTENCIA> ::= ; <SENTENCIA> <OTRA_SENTENCIA>
<OTRA_SENTENCIA> ::= <VACIO>
<SENTENCIA> ::= <ASIGNACION>
<SENTENCIA> ::= <LECTURA>
<SENTENCIA> ::= <ESCRITURA>
<ASIGNACION> ::= <VARIABLE> = <EXPRESION>
<EXPRESION> ::= <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= + <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= - <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= <VACIO>
<TERMINO> ::= <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= * <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= / <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= % <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= <VACIO>
<FACTOR> ::= <PRIMARIO> <EXP>
<EXP> ::= ^ <FACTOR>
<EXP> ::= <VACIO>
<PRIMARIO> ::= - <PRIMARIO>
<PRIMARIO> ::= <ELEMENTO>
<ELEMENTO> ::= ( <EXPRESION> )
<ELEMENTO> ::= <VARIABLE>
<ELEMENTO> ::= <CONSTANTE>
<LECTURA> ::= R <VARIABLE>
<ESCRITURA> ::= W <VARIABLE>
<CONSTANTE> ::= digito
<VARIABLE> ::= letra_minuscula
<VACIO> ::=
```

Tan sólo nos queda por definir la semántica del operador potenciación [WILH92, pp. 379 edición francesa]. Así en función de los tipos de los operandos se obtiene el tipo de resultado. También influye el signo de los operandos y si es igual o distinto de cero. En el caso de MUSIM/1 sólo hay tipos enteros, pero habrán de tenenerse en cuenta estas reglas para versiones futuras de MUSIM con más tipos.

Operador	1er operando	2º operando	tipo de resultado
^	int|float	int>0	igual al 1er operando
	int|float<>0	int=0	igual al 1er operando
	int|float<>0	int<0	float
	int|float>0	float	float
	int|float=0	float>0	float=0

Tabla 3: Semántica del operador potenciación

Ejercicio 19.6: Compilador ascendente de MUSIM/1 usando yacc

Se desea construir un compilador ascendente de MUSIM/1 a ENSAMPOCO/1 utilizando *yacc*. El lenguaje MUSIM/1 es el lenguaje MUSIM/0 con las siguientes ampliaciones: operador menos unario (-) y operador potenciación (^). También se ha ampliado el repertorio de instrucciones de ENSAMPOCO/0 añadiendo la instrucción EXP, que saca dos enteros de la pila de los cuales el primero es el exponente y el segundo la base, una vez ejecutada la instrucción se introduce en la pila el resultado de elevar la base al exponente.

Solución: En primer lugar la inclusión de dos operadores nos obliga a redefinir la tabla de precedencia de operadores. Se elige la solución más parecida a la que incorpora el lenguaje matemático. En este caso se ha realizado una pequeña variante en la cual la asignación está definida por medio del operador de asignación (=), al que se le asocia la precedencia más baja.

1ª `( )`
2ª `^`
3ª `- unario`
4ª `* / %`
5ª `+ -`
6ª `=`

A continuación se presenta el fichero completo de entrada al generador *yacc*, denominado *cmusim1.y*. El *yacc* y sus derivados *pcyacc®*, *yaccov*, *Bison*, etc... trabajan en modo línea de la forma siguiente:

```
yacc cmusim1.y
```

el resultado es el fichero *cmusim1.c*, que compilado con un compilador de C nos dará el ejecutable del compilador.

Un fichero de entrada al *yacc* tiene tres partes separadas por el símbolo %%.

- La primera parte son definiciones: tokens o símbolos terminales, símbolos no terminales y precedencia de operadores. También puede haber una sección de definiciones C o C++ que se incluye entre %{ y %} y que es código fuente en C que el generado colocará en la cabeza del programa generado.
- La segunda parte contien a la gramática que describe sintácticamente al lenguaje y entre llaves las acciones asociadas a cada regla de la gramática.
- La tercera parte son funciones o métodos escritos en lenguaje C o C++.

Los componentes léxicos o tokens se definen con el símbolo <*token*> indicando su tipo. Los operadores se ordenan de menor a mayor precedencia indicando su asociatividad por la izquierda, derecha, o no asociativos.

La sintaxis se especifica con una gramática con una notación parecida a la BNF pero indicando con minúsculas los símbolos no terminales, y con mayúsculas los símbolos terminales. Los símbolos terminales de un carácter también pueden aparecer en las reglas. Cada regla debe ir separada de la siguiente por un punto y coma (;). La parte derecha de cada regla se coloca después del símbolo dos puntos (:). Para indicar alternativa se utiliza el carácter barra vertical (|). El símbolo inicial de la gramática se define con la directiva *%start*. Por defecto se supone que el primer símbolo que aparece en la primera regla es el símbolo inicial.

Después de cada regla sintáctica están las acciones entre llaves, que es código fuente en C o C++. Las acciones realizan comprobaciones semánticas y generacón de código. En este caso particular tan sólo hay generación de código.

Debe observarse que las variables y funciones que comienzan con yy son las definidas de forma estándar por *yacc*.

La comunicación entre el analizador léxico y el sintáctico se realiza por lo que devuelve la función asociada al analizador léxico y por medio de la variable predefinida *yylval* que del tipo *%union* definido en cabeza.

El tratamiento de errores tratará de señalar todos los errores del programa fuente, y no como en el caso del compilador de MUSIM/0 que cada vez que determinaba un error paraba la compilación.

```
%{
/* cmusim1.y */
```

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <mem.h>
int nlinea=1;  /* contador del n° de líneas */
FILE *yyin,*yyout; /* Entrada y salida */
void yyerror(char*);
int yyparse(void);
void genera(char*);
void genera2(char*, int);
%}
%union
 { char identificador;
   int constante;
 }
%token  <constante> NUM
%token <identificador> VAR
%type <constante> bloque sentencia lectura escritura asignacion
%right '='
%left '-' '+'
%left '*' '/' '%'
%left NEG  /* Para darle precedencia al menos unario. */
%right '^'
%start programa
%%
programa     : 'M' '{' bloque finBloque
               ;
bloque       : { genera(".CODE"); }  instrucciones
               ;

finBloque    : '}'    {genera("END");}
               ;
instrucciones : /* vacío */
                | instrucciones sentencia
                | instrucciones sentencia ';'
                ;
sentencia   : lectura
              | escritura
              | asignacion
              ;
lectura     : 'R' VAR         { genera2("INPUT", $2);  }
              ;
escritura   : 'W' VAR         { genera2("OUTPUT", $2); }
              ;
asignacion  : VAR             { genera2("PUSHA", $1); }
               '=' exp         { genera("STORE");      }
              ;
exp         : NUM               {  genera2("PUSHC", $1); }
              | VAR               {  genera2("PUSHA", $1);
                                     genera("LOAD");       }
              | exp '+' exp       {  genera("ADD");        }
              | exp '-' exp       {  genera("NEG");
                                     genera("ADD");        }
              | exp '*' exp       {  genera("MUL");        }
              | exp '/' exp       {  genera("DIV");        }
              | exp '%' exp       {  genera("MOD");        }
              | '-' exp %prec NEG {  genera("NEG");        }
              | exp '^' exp       {  genera("EXP");        }
              | '(' exp ')'
              ;
%%
/*  analizador léxico  */
#include <ctype.h>
int yylex (void)
{
  int c;
```

```
  while ((c = getc(yyin)) != EOF)
   {
    if (c == ' '| c == '\t') continue; /* Elimina blancos y tabuladores */
    if (c == '\n')
      {++nlinea;
       continue;  /* Elimina los caracteres fin de línea */
      }
    if (isdigit (c))
     {
       yylval.constante = c;
       return NUM;
     }
    if (islower(c))
     {
      yylval.identificador = c;
      return VAR;
     }
    return c;
   }
  return 0;
}

/*  principal  */
void main(int argc, char *argv[])
{
  if ((argc<3) ||  (argc>4))
   {
    printf("\nCompilador MUSIM1-ENSAMPOCO1 desarrollado con YACCOV");
    printf("\nForma de uso:");
    printf("\ncmusim0 fich_fuente fich_destino\n");
    exit(-1);
   }
  if ((yyin=fopen(argv[1],"r")) == NULL)
   {
    printf("\nNo se puede abrir el fichero -> %s\n", argv[1]);
    exit(-2);
   }
  if ((yyout=fopen(argv[2],"w")) == NULL)
   {
    printf("\nNo se puede abrir el fichero -> %s\n", argv[2]);
    exit(-3);
   }
  printf("\nTraduciendo ...\n");
  yyparse();
  fcloseall();
  exit(0);
}

/*  subrutina de error  */
void yyerror (char *s)
{
  fprintf (stdout,"Error en línea %d : %s\n", nlinea, s);
}
void genera(char *s)
{
 fprintf(yyout,"%s\n",s);
}
void genera2(char *s, int x)
{
 fprintf(yyout,"%s %c\n", s, x);
}
```

Ejercicio 19.7: Gramática de MUSIM/11

Modificar la gramática de MUSIM/1 para que se utilicen *tokens* de más de un carácter. Se denominarán MUSIM/1x las variantes de MUSIM que profundicen en aspectos léxicos; las MUSIM/2x son las versiones que avancen en aspectos sintácticos; MUSIM/3x las que añadan aspectos semánticos; MUSIM/4x añaden características para soportar subprogramas y recursividad, con especial incidencia en las tablas de símbolos; MUSIM/5x añaden aspectos relativos a la gestión de memoria, en especial la gestion de memoria heap.

Solución: Existen infinitas soluciones, a continuación se muestra una de ellas.

```
<PROGRAMA> ::= MAIN  "{"  <BLOQUE>  "}"
<BLOQUE> ::= <SENTENCIA> <OTRA_SENTENCIA>
<OTRA_SENTENCIA> ::= ; <SENTENCIA> <OTRA_SENTENCIA>
<OTRA_SENTENCIA> ::= <VACIO>
<SENTENCIA> ::= <ASIGNACION>
<SENTENCIA> ::= <LECTURA>
<SENTENCIA> ::= <ESCRITURA>
<ASIGNACION> ::= <VARIABLE> = <EXPRESION>
<EXPRESION> ::= <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= + <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= - <TERMINO> <MAS_TERMINOS>
<MAS_TERMINOS> ::= <VACIO>
<TERMINO> ::= <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= * <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= / <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= % <FACTOR> <MAS_FACTORES>
<MAS_FACTORES> ::= <VACIO>
<FACTOR> ::= <PRIMARIO> <EXP>
<EXP> ::= ^ <FACTOR>
<EXP> ::= <VACIO>
<PRIMARIO> ::= - <PRIMARIO>
<PRIMARIO> ::= <ELEMENTO>
<ELEMENTO> ::= ( <EXPRESION> )
<ELEMENTO> ::= <VARIABLE>
<ELEMENTO> ::= <CONSTANTE>
<LECTURA> ::= READ <VARIABLE>
<ESCRITURA> ::= WRITE <VARIABLE>
<CONSTANTE> ::= entera
<VARIABLE> ::= letra_minuscula
<VACIO> ::=
```

Ejercicio 19.8: Gramática BNF de MUSIM/31

Construir la gramática en BNF del lenguaje MUSIM/31, resultante de ampliar el lenguaje MUSIM/11 con declaraciones obligatorias de variables de tipo entero (INTEGER), real (FLOAT), y carácter (CHARACTER). Las declaraciones se realizan en cabeza de programa colocando el identificador de tipo (INT, FLOAT, CHAR) y a continuación el nombre de variable, seguido de punto y coma (;). La gramática debe ser LL(1). Escribir ejemplos de programas correctos e incorrectos en lenguaje MUSIM/31.

Solución: Existen infinitas gramáticas que describen el lenguaje MUSIM/31, una de ellas que es LL(1) es la siguiente:

```
<Programa> ::= MAIN "{" <Bloque> "}"
<Bloque> ::= <Declaraciones> <Sentencias>
<Declaraciones> ::= <Declara> ; <Otra_declara>
<Otra_declara> ::= <Declara> ; <Otra_declara>
<Otra_declara> ::=  <Vacio>
<Declara> ::= <Tipo> <Variable>
<Tipo> ::= INTEGER
<Tipo> ::=  FLOAT
<Tipo> ::=  CHAR
<Sentencias> ::= <Instruccion> <Otra_instruccion>
<Otra_instruccion> ::= ; <Instruccion> <Otra_instruccion>
<Otra_instruccion> ::=  <Vacio>
<Instruccion> ::= <Asignacion>
<Instruccion> ::=  <Lectura>
<Instruccion> ::=  <Escritura>
<Asignacion> ::= <Variable> = <Expresion>
<Expresion> ::= <Termino> <Mas_terminos>
<Mas_terminos> ::= + <Termino> <Mas_terminos>
```

```
<Mas_terminos> ::=  - <Termino> <Mas_terminos>
<Mas_terminos> ::=  <Vacio>
<Termino> ::= <Factor> <Mas_factores>
<Mas_factores> ::= * <Factor> <Mas_factores>
<Mas_factores> ::=  / <Factor> <Mas_factores>
<Mas_factores> ::=  <Vacio>
<Factor> ::= <Primario> <Exp>
<Exp> ::= ^ <Factor>
<Exp> ::= <Vacio>
<Primario> ::= - <Primario>
<Primario> ::=  <Elemento>
<Elemento> ::= ( <Expresion> )
<Elemento> ::=  <Variable>
<Elemento> ::=  <Constante>
<Lectura> ::= READ <Variable>
<Escritura> ::= WRITE <Variable>
<Variable> ::= letra_minuscula
<Constante> ::= entera | real | caracter
<Vacio> ::=
```

Ejercicio 19.9: Semántica de MUSIM/31

La definición léxica y sintáctica de MUSIM/31 no es suficiente es necesario especificar su semántica. La semántica de este lenguaje está basada en las operaciones con sus tipos de datos.

Solución: Dada una especificación léxica y sintactica de un lenguaje la especificación semántica se puede orientar por diferentes caminos, aquí se ha elegido uno de ellos, una especificación semántica de comprobación estricta de tipos. Las promociones de tipo son muy restringidas en este caso se permite promocionar de entero a real. Los descensos de tipo están totalmente prohibidos, así por ejemplo está prohibido asignar directamente un real a un entero. La especificación semántica se puede resumir en:

- *Sistema de tipos*: Se permiten tres tipos de datos: CHAR (carácter), INTEGER (entero) y FLOAT (real). Por defecto todos los operadores aritméticos siguen las leyes de composición interna, es decir las operaciones que tienen operandos del mismo tipo tipo de datos, devuelven como resultado dicho tipos de datos, un caso especial es el operador potenciación (véase tabla 3):

  $$\text{<expresión>}_1\text{.tipo OPERADOR <expresión>}_2\text{.tipo} \rightarrow \text{<expresión>}_3\text{.tipo}$$

  La sentencia de asignación sigue la misma ley de composición interna, es decir el tipo de la variable de la parte izquierda debe ser igual al tipo de la expresión de la parte derecha.

  <variable>.tipo = <expresión>.tipo

- *Conversiones de tipo por promoción*: Tan solo se permite la promoción de tipo entero (INTEGER) a tipo real (FLOAT), es decir:

  $$\text{<expresión>}_1\text{.I OPERADOR <expresión>}_2\text{.F} \rightarrow \text{<expresión>}_3\text{.F}$$

  $$\text{<expresión>}_1\text{.F OPERADOR <expresión>}_2\text{.I} \rightarrow \text{<expresión>}_3\text{.F}$$

  También se permite sólo la promoción de entero (I) a real (F) en la sentencia de asignación:

  <variable>.F = <expresión>.I

- *Conversiones de tipo por descenso*: MUSIM/31 las prohibe totalmente. Por ejemplo no se permite pasar directamente de real a entero.

20. EJERCICIOS PROPUESTOS

Ejercicio 20.1: Arbol sintáctico de MUSIM/0

Sea la sentencia de MUSIM/0:

```
x=(-2*a+d*(b*b-4*a*c))/(2*a)
```

construir su árbol sintáctico de forma descendente, paso a paso a partir de expresión y expandiendo los símbolos no terminales comenzando siempre por el situado más a la izquierda.

Ejercicio 20.2: Comprobaciones semánticas de programas en MUSIM/31

Sean los programa de MUSIM/31 siguientes. Indicar si son correctos o no y si no lo son el motivo.

```
MAIN {
FLOAT a;
CHAR b;
READ a;
b=-a;
WRITE b }
```

```
MAIN {
INTEGER a,b;
FLOAT c,d;
READ a;
READ b;
c=a+b;
d=a-b;
WRITE c;
WRITE d}
```

```
MAIN {
INTEGER a;
FLOAT b;
FLOAT d
READ b; READ d;
a=b+d;
WRITE a}
```

Solución: 1) Error semántico en b=-a. 2) Correcto. 3) Error semántico en a=b+d

Ejercicio 20.3: Gramática de MUSIM/32

Escribir una gramática EBNF del lenguaje MUSIM/32, que es una ampliación del lenguaje MUSIM/31 con las especificaciones que se indicarán a continuación. La gramática se debe diseñar de una de las dos maneras siguientes: a) de forma que sea LL(1) b) de forma que esté preparada para un análisis sintáctico ascendente (por ejemplo para construir una entrada al *yacc*). Los tokens siguen siendo de un carácter.

- *Nuevo formato de declaración de variables*. Se introduce el separador coma (,) que permite la declaración de varias variables de un mismo tipo. A continuación se presenta un programa con declaraciones de MUSIM/32 válidas.

```
INTEGER a;
INTEGER b,c,d;
FLOAT e,f,g;
INTEGER x;
FLOAT y;
FLOAT u,z,t;
CHAR h,i;
```

- *Comentarios*. Un comentario en MUSIM/32 comienza por \ y acaba en fin de línea. A continuación se presenta un fragmento de programa en MUSIM/32 con comentarios válidos.

```
\Comentario en cabeza
MAIN {
\ Programa en MUSIM/32 con comentarios
INTEGER a; \ declaración de la variable a
FLOAT b,c;
READ a;READ b;READ c; \ Lectura de a,b y c
\ Aquí pueden ir los cálculos
WRITE a;WRITE b;WRITE c}
```

Solución: Para escribir declaraciones separadas por comas se puede escribir gramáticas BNF de la forma recursiva siguiente:

<declara> ::= <tipo> <variable> <otra_variable>
<otra_variable> ::= , <variable> <otra_variable>
<otra_variable> ::= <vacio>
<vacio> ::=

Los comentarios pueden ser eliminados en el análisis léxico, con lo cual no es necesario incluirlos en la gramática BNF que se utilizará para el análisis sintáctico.

Ejercicio 20.4: Gramática de PIPO/2

Ampliar la gramática del lenguaje PIPO, para lograr una calculadora científica que maneje además complejos denominada PIPO/2. Los complejos serán números escritos con el formato *34.2r35.1i*, para indicar el complejo con la parte real 34.2 y la parte imaginaria 35.1. El valor de tipo complejo debe escribirse todo unido sin espacios en blanco. También se pueden meter signos a las partes real e imaginaria, por ejemplo: *-34.2r-35.1i*. Las operaciones son las mismas, pero se añade la operación conjugado del complejo (cambio de signo de la parte imaginaria), que se representará por la letra *j*, y se aplicará de forma postfija al complejo situado en el tope de la pila. Obsérvese que se añade bastante complejidad a la calculadora al tener un nuevo tipo de datos.

Ejercicio 20.5: Gramática de MUSIM/33

Escribir una gramática EBNF del lenguaje MUSIM/32, que es una ampliación del lenguaje MUSIM/32 con las siguientes instrucciones y operadores.

- *Nuevo formato de instrucciones de lectura y escritura*. Se introduce el separador (,) que permite la lectura y escritura de varias variables de un mismo tipo separadas por un espacio en blanco. A continuación se presenta un fragmento de programa con sentencias de lectura y escritura de MUSIM/3 válidas.

```
MAIN{
INTEGER a,b;
FLOAT c,d;
CHAR e;
READ a, b, c;
READ d,e;
WRITE c,d,e,e,e }
```

- *Etiquetas de un dígito y bifurcación incondicional*. Se permite colocar etiquetas de un dígito que preceden a las sentencias, separadas de éstas por dos puntos (:). También se introduce la instrucción salto incondicional de la forma G <etiqueta>. Un ejemplo de programa con un bucle infinito es el siguiente:

```
\ Programa en MUSIM/33 con bucle infinito
MAIN {
FLOAT a,b,c;
3:READ a;READ b;
c=a+b;
WRITE c;
GOTO 3
}
```

- *Operadores de comparación*. Se introducen los operadores de comparación menor que (<), mayor que (>) y distinto que (#). Estos operadores se aplican sobre los tres tipos de datos y devuelven 0 (falso) y 1 (cierto). En el caso del tipo carácter se aplica el orden de la tabla ASCII. La precedencia de los operadores es:

  1ª ( )
  2ª ^
  3ª - unario
  4ª * / %
  5ª + -
  6ª < >
  7ª #

- *Sentencia alternativa simple*. Su sintaxis es de la forma siguiente:

```
IF <expresión> THEN <instrucción>;
```

  donde IF y THEN son dos nuevas palabras reservadas para este tipo de instrucción. La <expresión> puede cualquier tipo de expresión con cualquier número de operadores y paréntesis. Si la expresión es falsa, es decir vale 0, no se ejecuta la <instrucción>. Si la <expresión> toma un valor distinto de 0, si se ejecuta la <instrucción>. A continuación se muestra un ejemplo con la instrucción alternativa simple:

```
\Cálculo de la suma de n números y su media
MAIN {
FLOAT x,s, m;
INTEGER i,n;
READ n;
i=0;
s=0;
9:READ x;
s=s+x;
i=i+1;
IF i<n THEN GOTO 9;
m=s/i;
WRITE s, m }
```

Ejercicio 20.6: Gramática de MUSIM/34

Escribir una gramática EBNF del lenguaje MUSIM/34, que es una ampliación del lenguaje MUSIM/33 con las siguientes instrucciones y operadores.

a. Las instrucciones pueden ser simples o compuestas.

Se define sentencia compuesta como un conjunto de sentencias simples entre llaves ({ y }), y se permite el uso de sentencias compuestas donde anteriormente se permitían las instrucciones simples.

b. Operadores de manejo de booleanos

Los operadores son and (&), or (|) y not (!). Estos operadores se aplican sobre los tres tipos de datos, teniendo en cuenta que el valor 0 o carácter nulo es falso y el resto de los valores es cierto. La precedencia de operadores es:

1ª `( )`
2ª `^`
3ª `- unario !`
4ª `* / %`
5ª `+ -`
6ª `< >`
7ª `#`
8ª `&`
9ª `|`

Ejercicio 20.7: Gramática de MUSIM/35

Escribir una gramática EBNF del lenguaje MUSIM/35, que es una ampliación del lenguaje MUSIM/34 con la incorporación del tipo array.

- *Declaración de arrays*. Los arrays se declaran indicando

  `<tipo> <nombre> [ <tamaño> ]`

  donde el tipo es un tipo simple (INTEGER, FLOAT, CHAR), el nombre sigue siendo un identificador de una letra, y el tamaño es un entero. Los índices del array irán de 0 a tamaño-1.

  Pueden declararse arrays de más de una dimensión aplicando consecutivamente la definición de array. A continuación se muestra un ejemplo de declaración de arrays:

  ```
  FLOAT a[10]; \Array de 10 reales
  CHAR b[20]; \cadena de caracteres, el último elemento será \0
  INTEGER c[10][10]; \ array bidimensional de 100 elementos
  ```

- *Uso de arrays*. Las variables de tipo array subídicadas, es decir seguidas de su índice o índices, se pueden utilizar en cualquier lugar donde aparecía una variable simple. Con las declaraciones del ejemplo anterior se muestran las siguientes sentencias:

  ```
  i=2;
  READ a[0];
  READ a[i]; \i es una variable de tipo entero
  a[1]=a[i]+2;
  WRITE a[i];
  READ i, j, c[j][k];
  WRITE c[j][k];
  ```

- *No se permiten operaciones con arrays completos*. No se permite asignar un array a otro array, ni operar con un array completo con los operadores suma, resta, etc... (se controlorá por medio de acciones semánticas)

Ejercicio 20.8: Gramática de MUSIM/36

Escribir una gramática EBNF del lenguaje MUSIM/36, que es una ampliación del lenguaje MUSIM/35 con la incorporación de estructuras y uniones.

- *Declaración de estructuras y uniones*. En primer lugar es obligatorio declarar el tipo estructura con el formato:

  `STRUCT "{" { <tipo> <miembro> ; } "}" <nombre_tipo_struct>`

  Los tipos unión se declaran con el formato:

  `UNION "{" { <tipo> <miembro> ; } "}" <nombre_tipo_union>`

  El nombre del tipo estructura o unión sigue siendo un identificador de un dígito. Se permiten estructuras anidadas, es decir se permiten miembros de tipo estructura.

  Las variables de tipo estructura o union se declaran colocando el tipo declarado anteriormente seguido del identificador de la variable.

  Para el acceso a los miembros se utiliza el nombre de la variable estructura seguida de un punto y el nombre del miembro.

```
\Ejemplo
STRUCT { FLOAT a; INTEGER b; CHAR c[10]; } d; \declaración del tipo d
UNION { FLOAT e; INTEGER f; CHAR h; } i;     \declaración del tipo i
STRUCT { d l; FLOAT m; } n;
d j; \declaración de la variable j de tipo d
i k; \declaración de la variable k de tipo i
n o, p; \declaración de la variable o de tipo n
n r[10] ; array de 10 estructuras
...
READ o.m;
READ o.l.a;
p = o;
...
 También se permite declarar:
STRUCT {d x,y; FLOAT z} u;
```

- *Uso de estructuras y uniones*. Se utilizará el . para acceder a los miembros de la estructura. En el caso de estructuras anidadas se deberá aplicar consecutivamente el operador punto (.).
- *Se permiten la asignación con estructuras y uniones completas*. La única operación permitida con estructuras y uniones que las maneja en su totalidad es la asignación.

Ejercicio 20.9: Gramática del lenguaje LISP

Escribir la gramática del lenguaje de programación LISP.

Ejercicio 20.10: Gramática del lenguaje FORTRAN

Escribir la gramática del lenguaje de programación FORTRAN.

Ejercicio 20.11: Gramática del lenguaje COBOL

Escribir la gramática del lenguaje de programación COBOL.

Ejercicio 20.12: Gramática del lenguaje C

Escribir la gramática del lenguaje de programación C.

Ejercicio 20.13: Gramática del lenguaje Smalltalk

Escribir la gramática del lenguaje de programación Smalltalk.

Ejercicio 20.14: Gramática del lenguaje PROLOG

Escribir la gramática del lenguaje de programación PROLOG.

Ejercicio 20.15: Gramática del lenguaje C++

Escribir la gramática del lenguaje de programación C++.

Ejercicio 20.16: Gramática del lenguaje Eiffel

Escribir la gramática del lenguaje de programación Eiffel.

Ejercicio 20.17: Gramática del lenguaje Java

Escribir la gramática del lenguaje de programación Java.

21. PRACTICAS DE LABORATORIO

Ejercicio 21.1: Emulador software de PILAREGI

Construir en C++ un emulador software de la máquina PILAREGI siguiendo la metodología mostrada en el intérprete de ENSAMPOCO/0. El emulador debe ser transportable entre cualquier máquina y sistema operativo (ver anexo XVIII).

Ejercicio 21.2: Traductor de ENSAMPIRE a ensamblador 80x86

Siguiendo la metodología del traductor de ENSAMPOCO/0 a ensamblador 80x86, desarrollar en C++ un traductor de ENSAMPIRE a ensamblador 80x86 (ver anexo XVIII).

Ejercicio 21.3: Traductor directo de ENSAMPIRE a formato .COM de DOS

Siguiendo la metodología del traductor de ENSAMPOCO/0 a ensamblador 80x86, desarrollar en C++ un traductor de ENSAMPIRE a formato .COM de DOS. Directamente se generaran los ficheros a binario.

Ejercicio 21.4: Traductor directo de ENSAMPIRE a formato .OBJ de DOS

Siguiendo la metodología del traductor de ENSAMPOCO/0 a ensamblador 80x86, desarrollar en C++ un traductor de ENSAMPIRE a formato .OBJ de DOS. Directamente se generaran los ficheros a binario.

Ejercicio 21.5: Compilador de MUSIM/32 a ENSAMPIRE

Escribir en C++ un compilador de MUSIM/32 a ENSAMPIRE siguiendo la metodología del traductor de MUSIM/0 a ENSAMPOCO/0. Puede hacerse por dos caminos descendente o ascendente.

Ejercicio 21.6: Compilador de MUSIM/33 a ENSAMPIRE

Escribir en C++ un compilador de MUSIM/33 a ENSAMPIRE siguiendo la metodología del traductor de MUSIM/0 a ENSAMPOCO/0. Puede hacerse por dos caminos descendente o ascendente.

Ejercicio 21.7: Compilador de MUSIM/34 a ENSAMPIRE

Escribir en C++ un compilador de MUSIM/34 a ENSAMPIRE siguiendo la metodología del traductor de MUSIM/0 a ENSAMPOCO/0. Puede hacerse por dos caminos descendente o ascendente.

Ejercicio 21.8: Compilador de MUSIM/35 a ENSAMPIRE

Escribir en C++ un compilador de MUSIM/35 a ENSAMPIRE siguiendo la metodología del traductor de MUSIM/0 a ENSAMPOCO/0. Puede hacerse por dos caminos descendente o ascendente.

Ejercicio 21.9: Compilador de MUSIM/36 a ENSAMPIRE

Escribir en C++ un compilador de MUSIM/36 a ENSAMPIRE siguiendo la metodología del traductor de MUSIM/0 a ENSAMPOCO/0. Puede hacerse por dos caminos descendente o ascendente.

Ejercicio 21.10: Compilador de MUSIM/37 a ENSAMPIRE

Escribir en C++ un compilador de MUSIM/37 a ENSAMPIRE siguiendo la metodología del traductor de MUSIM/0 a ENSAMPOCO/0. Puede hacerse por dos caminos descendente o ascendente. MUSIM/37 incorpora tipos objeto al estilo de Object Pascal, permitiendo encapsulación, herencia simple y polimorfismo.

Ejercicio 21.11: Entorno integrado del compilador MUSIM

Escribir un entorno integrado para los compiladores MUSIM que incluya: editor con sintaxis resaltada de colores, depurador, traductor a lenguaje intermedio ENSAMPIRE, generador de código ensamblador 80x86, generador de ejecutables .COM o .EXE para DOS. También se deben incluir las pruebas de validación y la documentación.

ANEXO I El lenguaje máquina 80x86

El lenguaje máquina está directamente ligado a la arquitectura de los ordenadores de tal forma que un programa en código máquina será solamente válido para un tipo o familia de máquinas y además debe de expresarse según un formato especificado por el sistema operativo. Un programa en lenguaje máquina siempre debe de indicar el tipo de ordenadores y sistema operativo para los cuales es válido.

Se presenta un programa *hola.com* de 24 bytes que escribe en pantalla *Hola a todos*. Esta desarrollado para microprocesadores de la familia 80x86 y para el sistema operativo DOS.

PROGRAMA EN LENGUAJE MAQUINA EN BINARIO
11101011 00001110 10010000 01001000 01101111 01101100 01100001 00100000 01100001 00100000 01110100 01101111 01100100 01101111 01110011 00100100 10111010 00000011 00000001 10110100 00001001 11001101 00100001 11000011

Tabla 4: Programa *hola.com* en binario

Por razones de eficacia y comodidad los programas en lenguaje máquina suelen escribirse en base 16, también se presenta el programa anterior en base 16, con unas explicaciones de su contenido en lenguaje ensamblador. El prograna *hola.com* ocupa 24 bytes.

Programa en código máquina en base 16	Explicación en ASM
EB 0E	JMP 0101
90	NOP
48 6F 6C 61 20 61 20 74 6F 64 6F 73 24	Hola a todos$
BA 03 01	MOV DX, 0103
B4 09	MOV AH,09
CD 21	INT 21H
C3	RET

Tabla 5: Programa *hola.com* en base 16

Sobre la estructura de los ficheros .OBJ y .COM consultar [GARC91, SIER91a, SIER91b].

ANEXO II El lenguaje ensamblador 80x86

El lenguaje ensamblador es una representación simbólica del lenguaje máquina, por lo tanto estrá también ligado a una familia de ordenadores. Sin embargo el lenguaje ensambldor permite al programador escabullirse de un conjunto de tareas y controles rutinarios que debería de realizar obligatoriamente en lenguaje máquina.

Se escribe un programa *hola.asm* que escribe en pantalla *Hola a todos*, para que el ordenador pueda procesarlo es necesario que un ensamblador lo traduzca en primer lugar a código máquina.

```
; Programa en lenguaje ensamblador 80x86 bajo sistema operativo DOS
SALUDO   SEGMENT    ;Establece un segmento que contiene datos y código
           ASSUME CS:SALUDO, DS:SALUDO
           ORG 100H   ; Un programa .COM
PRINCIPAL:JMP INICIO
MENSAJE DB 'Hola a todos$'
INICIO   :MOV DX, OFFSET MENSAJE ; asigna a DX la dirección de MENSAJE
          MOV AH,9               ; Función del DOS para escribir una cadena
          INT 21H                ; Llama al DOS
          RET                    ; Vuelve al sistema operativo
SALUDO   ENDS
         END PRINCIPAL
```

Tabla 6: Programa *hola.asm*

El programa del cuadro 3 se traduce a código máquina por medio de un ensamblador, por ejemplo MASM de Microsoft®, y se crea un fichero *hola.obj*. Por medio del montador de enlaces, LINK, se crea un fichero ejecutable *hola.exe*. En este caso el programa sólo utiliza un único segmento y se puede cargar directamente en memoria, es decir se puede crear con la utilidad EXE2BIN del DOS un fichero ejecutable *hola.com*. La representación en código máquina de este fichero *hola.com* es la que se muestra en las tablas 3 y 4. El tamaño de este fichero es de 24 bytes. También se pueden crear programas .COM con el TASM de Borland® y la opción /t.

Se pueden construir programas en ensamblador que tengan más de un segmento, así en la tabla 5 se presenta un programa *ejemasm.asm*, que escribe también el mensaje *Hola a todos* en pantalla. Utilizando un ensamblador (MASM o TASM) se crea el fichero *ejemasm.obj* y por medio del montador de enlaces (LINK o TLINK) se crea el fichero ejecutable *ejemasm.exe*, que tiene un tamaño de 610 bytes.

```
; Ejemplo de programa en ensamblador 80x86 bajo DOS para crear un .EXE
;
DATOS     SEGMENT PUBLIC 'DATA'
   SALUDO DB 'Hola a todos$'
DATOS     ENDS

PILA      SEGMENT STACK 'STACK'
          DB 64 DUP ('?')
PILA      ENDS

CODIGO    SEGMENT PUBLIC 'CODE'
          ASSUME CS:CODIGO, DS:DATOS
PRINCIPAL PROC FAR
INICIO:   PUSH DS         ; Guarda la posición del registro DS
          SUB AX,AX       ; Pone a cero AX
          PUSH AX         ; Introduce un cero en la pila
          MOV AX,DATOS    ; Guarda la posición de datos en AX
          MOV DS,AX       ; Copia la posición de datos en DS
;
; Aquí empieza el programa
;
          LEA DX, SALUDO
          MOV AH,9
          INT 21H
;
; Fin del programa
;
          RET     ; Devuelve el control al DOS
PRINCIPAL ENDP
CODIGO    ENDS
          END INICIO
```

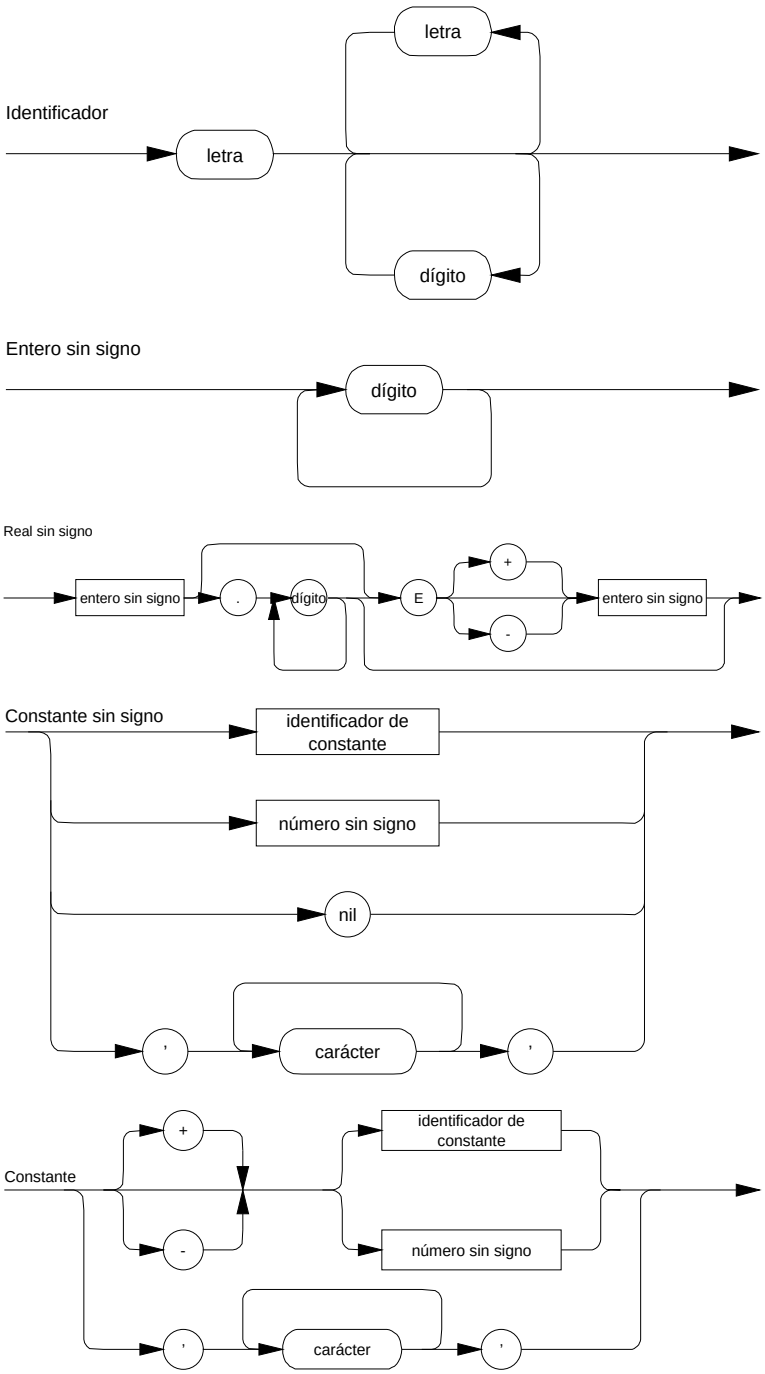
Tabla 7: Programa *ejemasm.asm*

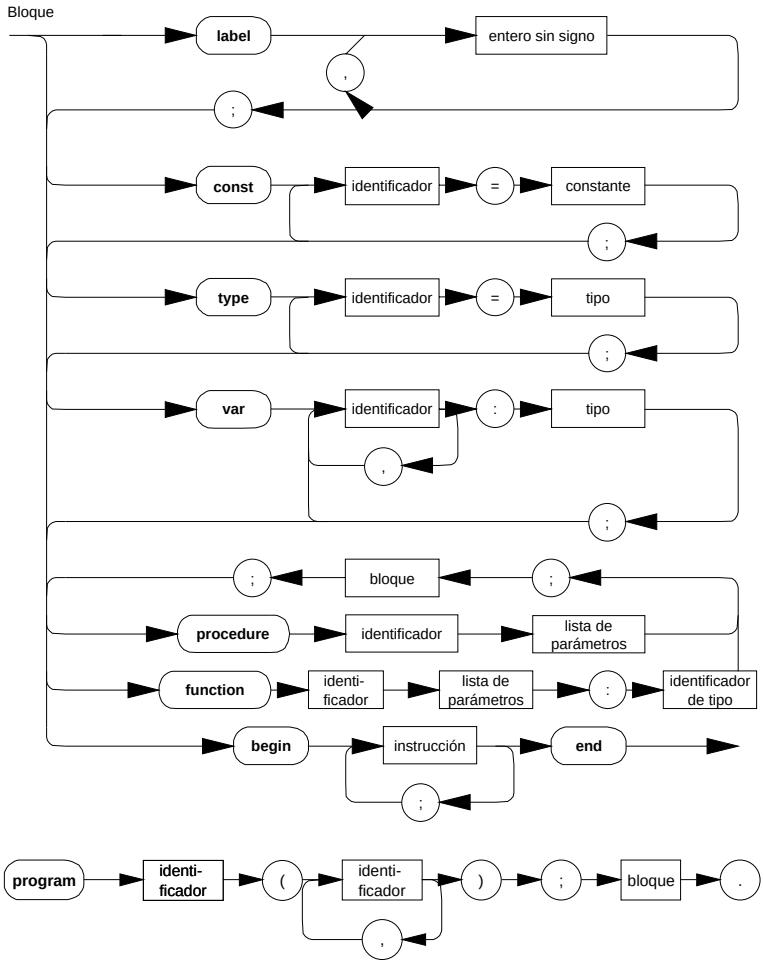
El ensamblador ASM90 [GARC91] ha sido desarrollada en la Universidad de Oviedo, construye ficheros .OBJ a partir de ficheros fuente .ASM. Tanto el LINK de Microsoft y el TLINK de Borland pueden utilizarse para el montaje de los módulos producidos por ASM90. El ASM90 también incorpora una biblioteca para ser utilizada desde dentro de compiladores, para que estos traduzcan directamente a .OBJ. Para más información puede consultarse [GARC91].

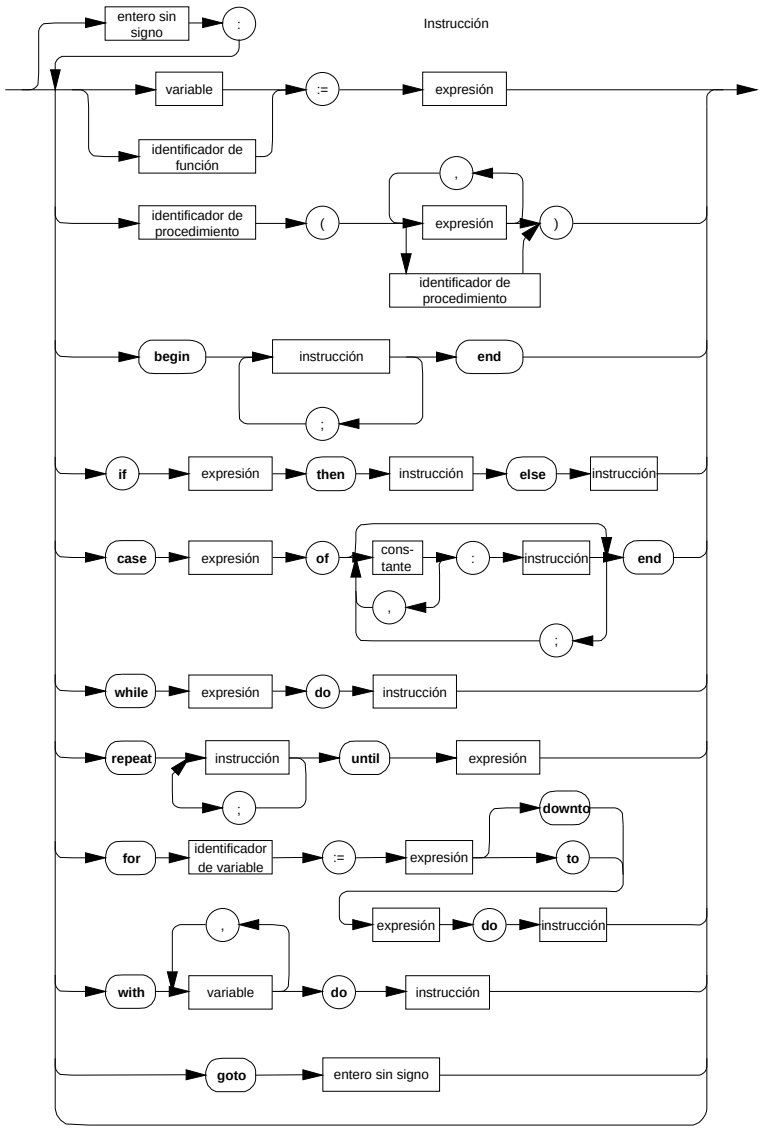
ANEXO III El lenguaje Pascal

III.1 Diagramas sintácticos

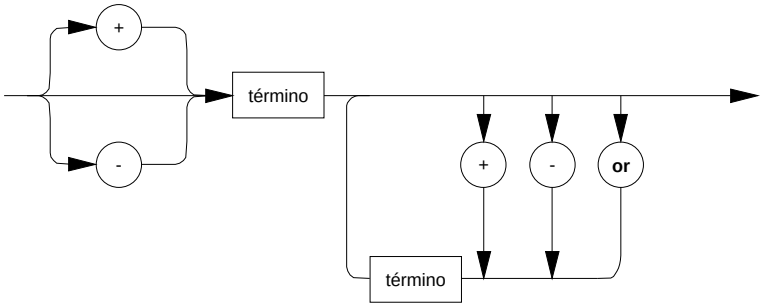
Estos diagramas constituyen parte de la definición del lenguaje Pascal, *Jensen K. y N. Wirth (1975).*



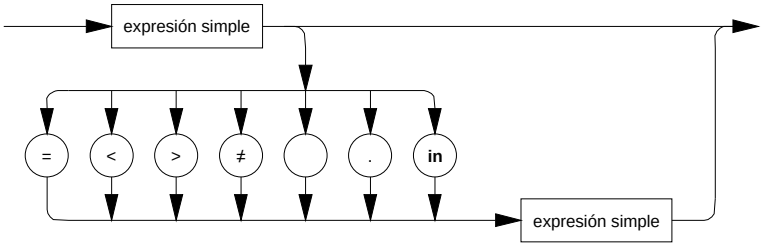

Bloque
label
entero sin signo
,
;
const
identificador
=
constante
;
type
identificador
=
tipo
;
var
identificador
:
tipo
,
;
;
bloque
;
procedure
identificador
lista de parámetros
function
identi-ficador
lista de parámetros
:
identificador de tipo
begin
instrucción
end
;
program
identi-ficador
(
identi-ficador
)
;
bloque
.
,



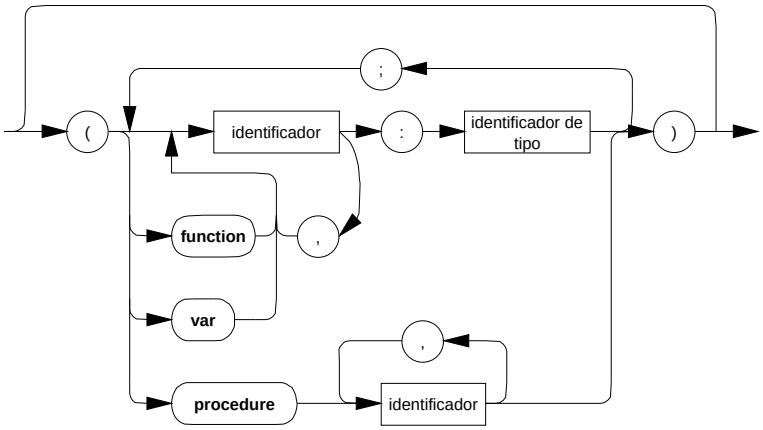
Expresión simple



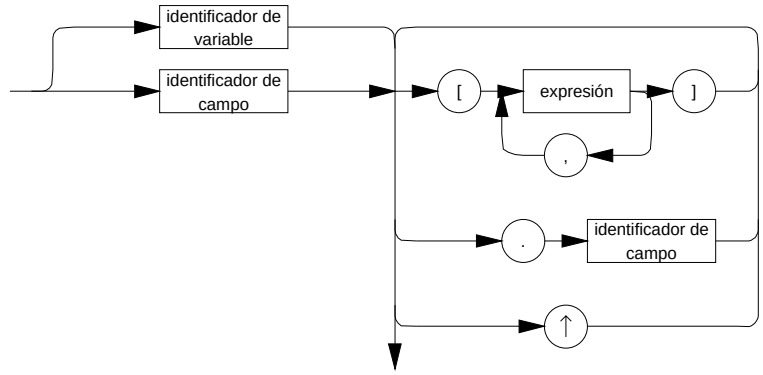
Expresión



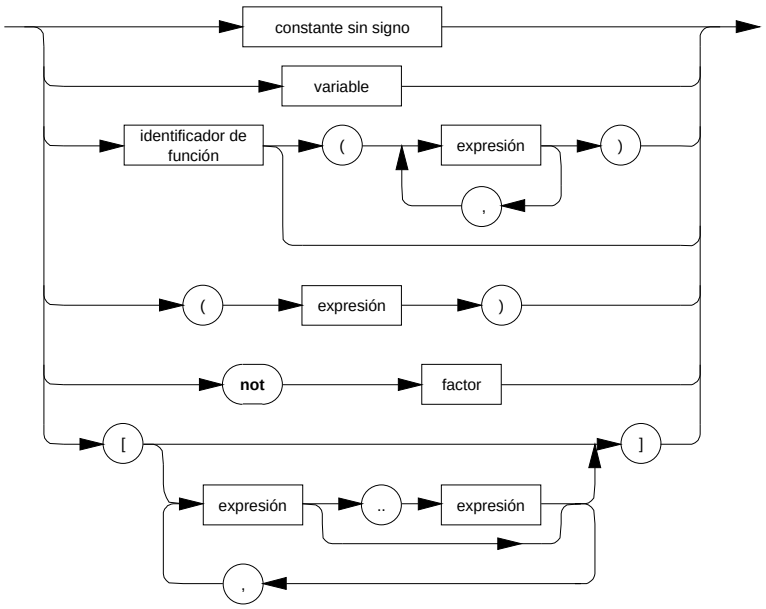
Lista de parámetros



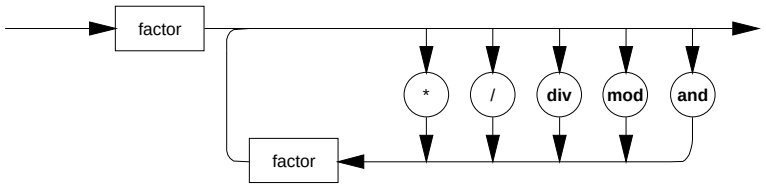
Variable



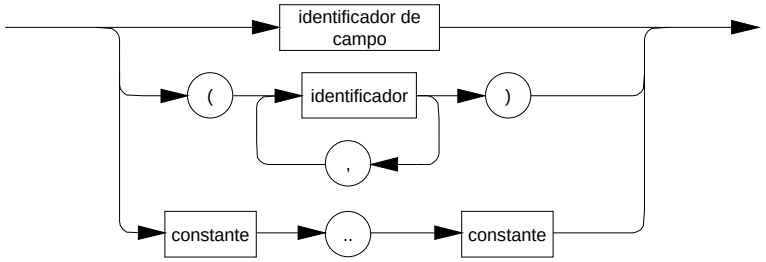
Factor



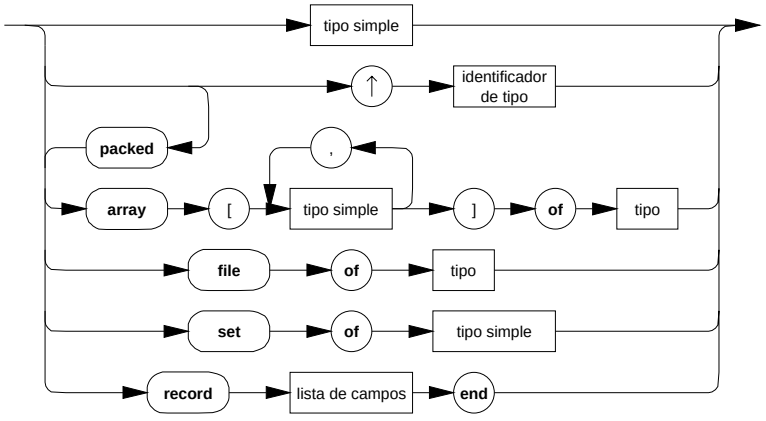
Término

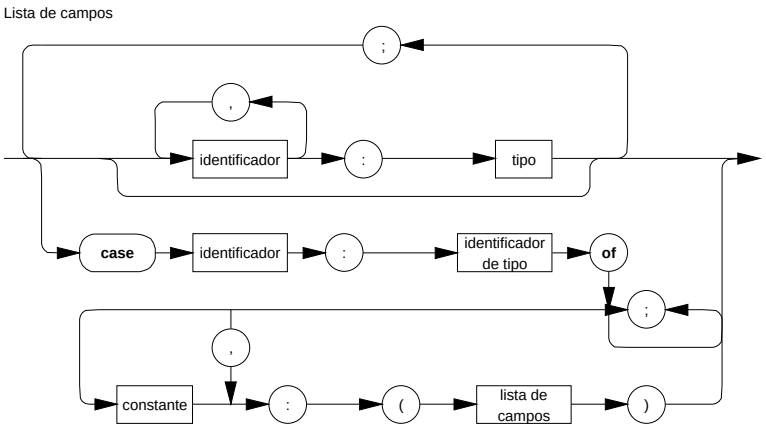


Tipo simple



Tipo





III.2 Gramática EBNF

Esta gramática es parte de la definición del lenguaje Pascal *Jensen K. y N. Wirth* [JENS75].

<programa>::= <encabezamiento del programa><bloque>.
<encabezamiento>::= *program* <identificador> (<identificador de archivo>
{, <identificador de archivo>)};
<identificador de archivo>::= <identificador>
<identificador>::= <letra> {<letra o dígito>}
<bloque>::= <parte de declaración de rótulos>
<parte de definición de constantes>
<parte de definición de tipos> <parte de declaración de variables>
<parte de declaración de procedimientos y funciones> <sentencias>
<parte de declaración de rótulos>::= <vacía> | *label* <rótulo>{, <rótulo>};
<rótulo>::= <entero sin signo>
<parte de definición de constantes>::= *const* <definición de constante>
{; <definición de constante>};
<definición de constante>::= <identificador>= <constante>
<constante>::= <número sin signo> | <signo><número sin signo> |
<identificador de constante> | <signo><identificador de constante> |
<cadena de caracteres>
<número sin signo>::= <entero sin signo> | <real sin signo>
<entero sin signo>::= <dígito> {<dígito>}
<real sin signo>::= <entero sin signo> . <dígito> {<dígito>} |
<entero sin signo> . <dígito> {<dígito>| E <factor de escala> |
<entero sin signo> E <factor de escala>
<factor de escala>::= <entero sin signo> | <signo><entero sin signo>
<signo>::= + | -
<identificador de constante>::= <identificador>
<cadena de caracteres>::= '<carácter> {<carácter>}'
<parte de definición de tipos>::= <vacía> |
type <definición de tipo> {; <definición de tipo>};
<definición de tipo>::= <identificador>= <tipo>
<tipo>::= <tipo simple> | <tipo estructurado> | <tipo puntero>
<tipo simple>::= <tipo escalar> | <tipo subrango> | <tipo identificador>
<tipo escalar>::= (<identificador>, {<identificador>})
<tipo subrango>::= <constante> .. <constante>
<tipo identificador>::= <identificador>
<tipo estructurado>::= <tipo estructurado no empaquetado> |
packed <tipo estructurado no empaquetado>
<tipo estructurado no empaquetado>::= <tipo arreglo> | <tipo registro> |
<tipo conjunto> | <tipo archivo>
<tipo arreglo>::= *array* [<tipo índice> {, <tipo índice>}] *of*
<tipo componente>
<tipo índice>::= <tipo simple>
<tipo componente>::= <tipo>
<tipo registro>::= *record* <lista de campos> *end*
<lista de campos>::= <parte fija> | <parte fija>; <parte variable>|
<parte variable>
<parte fija>::= <sección de registro> {; <sección de registro>}
<sección de registro>::= <identificador de campo>
{, <identificador de campo>}:
<tipo> | <vacío>
<parte variable>::= *case* <campo de etiquetas> <identificador de tipo> *of*
<variante> {; <variante>}
<campo de etiquetas>::= <identificador del campo>: | <vacío>
<variante>::= <lista de rótulos del case>: (<lista de campos>) | <vacío>
<lista de rótulos del case>::= <rótulo del case> {, <rótulo del case>}
<rótulo del case>::= <constante>
<tipo conjunto>::= *set of* <tipo base>
<tipo base>::= <tipo simple>

<tipo archivo>::= *file of* <tipo>
<tipo puntero>::= ↑<identificador de tipo>
<parte de declaración de variable>::= <vacía> |
var <declaración de variable> {; <declaración de variable>
<declaración de variable>::= <identificador> {, <identificador>}: <tipo>
<parte de declaración de procedimientos y funciones>::=
<declaración de procedimientos> | <declaración de funciones>
<declaración de procedimientos>::= <encabezamiento de procedimiento>
| <bloque>
<encabezamiento de procedimiento>::= *procedure* <identificador>; |
procedure <identificador> (<sección de parámetros formales>
{; <sección de parámetros formales>});
<sección de parámetros formales>::= <grupo de parámetros> |
var <grupo de parámetros> | *function* <grupo de parámetros> |
procedure <identificador> {, <identificador>}
<grupo de parámetros>::= <identificador> {, <identificador>}:
<identificador de tipo>
<declaración de función>::= <encabezamiento de función><bloque>
<encabezamiento de función>::= *function* <identificador> :
<tipo de resultado>; |
function <identificador> (<sección de parámetros formales>
{; <sección de parámetros formales>}): <tipo de resultado>;
<tipo de resultado>::= <identificador de tipo>
<parte de sentencias>::= <sentencia compuesta>
<sentencia>::= <sentencia sin rótulo> | <rótulo><sentencia sin rótulo>
<sentencia sin rótulo>::= <sentencia simple> | <sentencia estructurada>
<sentencia simple>::= <sentencia de asignación> |
<sentencia de procedimiento> |
<sentencia go to> | <sentencia vacía>
<sentencia de asignación>::= <variable>:= <expresión> |
<identificador de función>:= <expresión>
<variable>::= <variable completa> | <componente de variable> |
<referencia a variable>
<variable completa>::= <identificador de variable>
<identificador de variable>::= <identificador>
<componente de variable>::= <variable indexada> |
<designador de campo> | <buffer de archivo>
<variable indexada>::= <variable de arreglo> [<expresión> {, <expresión>}]
<variable de arreglo>::= <variable>
<designador de campo>::= <variable de registro> . <identificador de campo>
<variable de registro>::= <variable>
<identificador de campo>::= <identificador>
<buffer de archivo>::= <variable de archivo>↑
<variable de archivo>::= <variable>
<referencia a variable>::= <variable puntero>↑
<expresión>::= <expresión simple> | <expresión simple>
<operador relacional>
<expresión simple>
<operador relacional>::= = | <> | < | > | <= | >= | *in*
<expresión simple>::= <término> | <signo><término> |
<expresión simple><operador de adición><término>
<operador de adición>::= + | - | *or*
<término>::= <factor> | <término><operador de multiplicación><factor>
<operador de multiplicación>::= * | / | *div* | *mod* | *and*
<factor>::= <variable> | <constante sin signo> | (<expresión>) |
<designador de función> | <conjunto> | *not* <factor>
<constante sin signo>::= <numero sin signo> | <cadena de caracteres> |
<identificador de constantes> | *nil*
<designador de función>::= <identificador de función> |
<identificador de función>
(<parámetro actual> {, <parámetro actual>})
<identificador de función>::= <identificador>
<conjunto>::= { <lista de elementos> }
<lista de elementos>::= <elemento> {, <elemento>} | <vacía>
<elemento>::= <expresión> | <expresión> .. <expresión>
<sentencia de procedimiento>::= <identificador de procedimiento> |
<identificador de procedimiento> (<parámetro actual>
{, <parámetro actual>})
<identificador de procedimiento>::= <identificador>
<parámetro actual>::= <expresión> | <variable> |
<identificador de procedimiento> | <identificador de función>
<sentencia go to>::= *goto* <rótulo>
<sentencia vacía>::= <vacía>
<vacía>::=
<sentencia estructurada>::= <sentencia compuesta> |
<sentencia condicional> | <sentencias repetitivas> | <sentencia with>
<sentencia compuesta>::= *begin* <sentencia> {; <sentencia>} *end*
<sentencia condicional>::= <sentencia if> | <sentencia case>
<sentencia if>::= *if* <expresión> *then* <sentencia> |
if <expresión> *then* <sentencia> *else* <sentencia>
<sentencia case>::= *case* <expresión> *of* <elemento de la lista case>
{; <elemento de la lista case>} *end*

<elemento de la lista case>::= <lista de rótulos de case>: <sentencia> |
<vacía>
<lista de rótulos de case>::= <rótulo de case> {, <rótulo de case>}
<sentencias repetitivas>::= <sentencia while> | <sentencia repeat> |
<sentencia for>
<sentencia while>::= *while* <expresión> *do* <sentencia>
<sentencia repeat>::= *repeat* <sentencia> {; <sentencia>}
until <expresión>
<sentencia for>::= *for* <variable de control>:= <lista for> *do* <sentencia>
<lista for>::= <valor inicial> *to* <valor final> |
<valor inicial> *downto* <valor final>
<variable de control>::= <variable>
<valor inicial>::= <expresión>
<valor final>::= <expresión>
<sentencia with>::= *with* <lista de variables de registro> *do* <sentencia>
<lista de variables de registro>::= <variable de registro>
{, <variable de registro>}

III.3 Definición formal semántica

Sobre la definición formal de la semántica del lenguaje Pascal puede consultarse la obra de *R.D. Tennent* [TENN81, pp. 261].

ANEXO IV El lenguaje C

El lenguaje C es un lenguaje de medio nivel desarrollado en 1973 por *Brian W. Kernighan* y *Dennis M. Ritchie* [KERN78, KERN88] especializado en el desarrollo de sistemas operativos y compiladores.

En este ejemplo el programa *hola.c* escribe el mismo saludo que en los ejemplos anteriores, sin embargo puede observarse que el programador no tiene que ocuparse de las interrupciones del microprocesador que escriben en pantalla, ni de reservar espacio para la cadena que constituye el mensaje de saludo, el programador sólo tiene que indicar que escriba el mensaje en un lenguaje próximo al humano, en este caso en lengua inglesa.

```
/* Programa en C */
main(void)
{
 printf ("Hola a todos\n");
}
```

Tabla 8: Programa *hola.c*

Este programa se compila con un compilador de C, por ejemplo el de Borland®, y se obtiene un programa ejecutable *hola.exe*, que en el caso de la versión 3.0 de Borland es de 8631 bytes, que es muy superior a los 24 bytes que ocupaba el mismo programa *hola.com* escrito directamente en código máquina, y a los 610 bytes del fichero *ejemasm.exe* cuando se generó por medio del MASM a .EXE.

Las comodidades y facilidades del programador se *pagan* en el tamaño del fichero ejecutable, lo cual se traduce en un incremento del tiempo de ejecución del programa.

Sobre compiladores de C se puede consultar [HEND88, SALG89, GARM91].

ANEXO V El lenguaje Simula

La historia del lenguaje Simula comienza en los años cincuenta y sesenta, cuando dos científicos noruegos Kristen Nygaard y Ole-Johan Dahl, pretendian realizar la simulación de sistemas, y desearon definir la estructura y actividad de los sistemas a modelar. El primer lenguaje que desarrollarón se denominó Simula 1, y tenía muchos elementos comunes con el ALGOL 60 (estructura de bloques y procedimientos, ámbito de las variables,etc...). De hecho las primeras implementaciones de Simula se realizarón en ALGOL 60. El lenguaje Simula 1 se presentó públicamente en 1962, en una reunión en Munich de la IFIP. Los objetos y las clases aparecieron cinco años más tarde en el Simula 67. Hoare trabajó en el desarrollo del compilador, consiguiendo que la mayor parte de las comprobaciones se realizasen en tiempo de compilación.

Los principales desarrollos del lenguaje Simula 67 fueron:

- Introduce el concepto de encapsulación y objetos.
- Se advirtió que algunas clases de objetos pueden tener propiedades comunes, sugiriendo un esquema de factorización, denominado actualmente herencia.
- Introdujo el concepto de virtual y enlace dinámico (dynamic binding).
- Se comenzó con la simulación de la concurrencia.

El Simula 67 es un superconjunto del lenguaje ALGOL 60, al que se le añadieron nuevas características que constituyen el núcleo principal de lo que se denomina actualmente programación orientada a objetos. Se puede decir que probablemente todos los lenguajes de programación orientados a objetos descienden de alguna forma del Simula 67.

ANEXO VI El lenguaje Smalltalk

Fue creado por los miembros del Learning Research Group, pertenecientes al Xerox Palo Alto Research Center. En un principio Smalltalk era una parte del software del Dynabook, un proyecto futurista de Alan Kay. Smalltalk representa un lenguaje y un entorno de desarrollo de software, puede considerarse como un lenguaje orientado a objetos puro, en el que cualquier cosa se considera un objeto. Smalltalk se construyó alrededor de dos conceptos muy simples: cualquier cosa es un objeto, y los objetos se comunican entre sí por medio del paso de mensajes [GOLD83, GOLD89].

Smalltalk ha evolucionado durante una década, desde el Smalltalk-72 al Smalltalk-80. Actualmente se comercializa en distintos entornos, tanto a nivel de ordenadores personales, como en workstations. Quizá su principal inconveniente a nivel comercial, sea que es un lenguaje interpretado, dado que la mayor parte de las comprobaciones son dynamic binding y se realizan en tiempo de ejecución.

Después del lenguaje Simula 67, el lenguaje Smalltalk es quizá el lenguaje que más ha influído en los lenguajes orientados a objetos, y también en los GUI (*graphic user interfaces*) tales como Macintosh, X-Windows, Motif y MS-Windows. También se deben a Smalltalk los *browsers* (inspectores de objetos), las colecciones, y las clases contenedor.

ANEXO VII El lenguaje Ada

Fue desarrollado a propuesta del Department of Defense (DoD) de los Estados Unidos a finales de los años 70. Sus antecesores han sido Pascal y sus derivados (Euclid, Lis, Mesa, Modula, y Sue); así como ALGOL 68, Simula, CLU y Alphard. La norma ANSI se publicó en 1983. Los compiladores de Ada requieren un hardware potente y máquinas modestas son lentos, aunque en arquitecturas adecuadas son eficientes.

Ada es un lenguaje basado en objetos, y no un lenguaje orientado a objetos, aunque el borrador del Ada 95, incluye todas las características de un lenguaje orientado a objetos (encapsulación, herencia y polimorfismo).

ANEXO VIII El lenguaje C++

Fue diseñado por Bjarne Stroustrup de AT&T en 1985. Su antecesor es el lenguaje C con clases, también desarrollado por Stroustrup en 1980. Es una mezcla de los lenguajes C y Simula 67. Es un lenguaje híbrido en el que se prima la eficiencia, siguiendo la pauta de su antecesor el lenguaje C. Es el lenguaje orientado a objetos más utilizado en la actualidad a nivel comercial, debido en gran parte a su eficiencia y a las ampliaciones de sus últimas revisiones.

El lenguaje C++ es una ampliación y modificación del lenguaje C, para incorporarle tipos abstractos de datos, que reciben el nombre de clases. El lenguaje de programación fue creado por *Bjarne Stroustrup* de los laboratorios AT&T Bell, que lo popularizó con la publicación en 1986 del libro *The C++ programming language* [STRO86], en el que se establecieron las bases del lenguaje. Actualmente el comité ANSI X3J16 está normalizando el lenguaje, y admite como texto básico el libro de *Margaret Ellis y Bjarne Stroustrup* titulado *The anotated C++ reference manual* [ELLI90]. En 1991 *Bjarne Stroustrup* ha sacado la segunda versión y actualización del libro *The C++ programming language* [STRO91]. El mercado sigue básicamente la numeración AT&T, para distinguir entre las distintas versiones y actualizaciones. Así hasta este momento se han comercializado las versiones 1.2, 2.0, 2.1, y 3.0.

En el ejemplo *hola.cpp* (tabla 9) se vuelve escribir un programa que emite un saludo, pero esta vez utilizando las clases predefinidas en *iostream.h*, que permiten usar los manejadores de entrada y salida. Este programa compilado con Borland C++ versión 3.0, produce un programa ejecutable de 23136 bytes. Puede observarse un incremento considerable de tamaño respecto a la versión de C, en parte debido a la inclusión de las clases predefinidas.

```
// Programa hola.cpp
#include <iostream.h>
main(void)
{
   cout << "Hola a todos\n";
}
```

Tabla 9: Programa *hola.cpp* en C++

ANEXO IX El lenguaje Object Pascal

Fue creado por los desarrolladores de Apple Computer con N. Wirth (autor del Pascal). Su antecesor fue el Clascal, una versión orientada a objetos para Lisa (de Apple). Se presento en 1986 y fue el primer lenguaje soportado por el Macintosh Programmer's Workshop (MPW). La biblioteca de clases MacApp daba el marco de trabajo para construir aplicaciones en el entorno Macintosh.

Object Pascal ha inspirado la versión de Pascal orientado a objetos de Borland (Turbo Pascal, Delphi) y de Microsoft (Quick Pascal).

Todas las versiones son compiladas, y con ellas se han desarrollado aplicaciones eficientes. Sin embargo al ser un lenguaje híbrido, no dispone de diversos mecanismos de la programación orientada a objetos, por ejemplo la genericidad [CUEV94a].

ANEXO X El lenguaje CLOS (Common Lisp Object System)

Existen docenas de dialectos del lenguaje Lisp (Scheme, ZetaLisp, Nil, etc...). A principios de los 80 comienzan a aparecer nuevos dialectos de Lisp, que incorporan aspectos de la programación orientada a objetos. En 1986, después de la normalización del Lisp, en el denominado Common Lisp, se crea un grupo de trabajo especial para la normalización de un superconjunto del Common Lisp con extensiones del lenguaje orientadas a objetos, al nuevo lenguaje se le denomina con las siglas CLOS (Common Lisp Object System).

El diseño del CLOS fue influenciado por los dialectos del Lisp y Smalltalk denominados New Flavors y CommonLoops. La especificación completa de CLOS se publicó en 1988.

El lenguaje CLOS es interpretado, y existen implementaciones para diversas plataformas [PAEP93].

ANEXO XI El lenguaje Eiffel

El lenguaje Eiffel ha sido desarrollado por Bertrand Meyer con el propósito de obtener un lenguaje orientado a objetos puro, cumpliendo las características del cuadro 1. Las ideas más notables de Eiffel proceden de Simula 67, ALGOL y lenguajes derivados, Alphard, CLU, Ada y el lenguaje de especificación Z.

Las implementaciones existentes de lenguaje Eiffel son traductores a lenguaje C, y en general son muy ineficientes. Por otra parte la característica de gestión de memoria automática, implementada con recolectores de basura (garbage collection) suponen un límite a su uso como lenguaje en tiempo real.

Sin embargo las aportaciones de Eiffel como lenguaje pueden marcar las líneas maestras del diseño de los lenguajes orientados a objetos del futuro [MEYE88, MEYE92].

ANEXO XII El lenguaje Postscript®

Postscript® es el nombre de un lenguaje de programación desarrollado por ADOBE System Incorporated para enviar información gráfica de alto nivel a periféricos *raster*, por ejemplo impresoras láser, filmadoras, etc... Este lenguaje está especializado en la composición de páginas.

Como lenguaje de programación destacar que no tiene palabras reservadas, el número de caracteres especiales es muy pequeño, es de notación postfija, y está concebido para que sus programas sean generados por otro software. Los programas se escriben con los caracteres ASCII, esto permite que sean tratados como ficheros de texto y que su lectura sea tan fácil como la estructura del programa lo permita.

El lenguaje es interpretado. Mediante un analizador léxico se agrupan los caracteres para formar objetos, éstos son interpretados según su tipo y atributos. El significado de los operadores y procedimientos no se conoce hasta que el intérprete lo encuentra y ejecuta; su significado puede cambiar a lo largo del programa.

El lenguaje contiene tipos de datos comunes a otros lenguajes de programación, tales como *boolean*, *reales*, *enteros*, *arrays*, *ficheros*, y *strings*. Y otros tipos especiales como son *dictionary*, *names*, *mark*, *save*, *null*, *operadores* y *FontID*. Los diccionarios junto con los nombres forman un conjunto que aporta gran potencia al lenguaje, con ellos se pueden simular las variables y se le da nombre a los procedimientos para que puedan ser llamados.

El lenguaje controla cuatro pilas con las que mantiene el entorno en el cual se ejecuta el programa. La más importante es la *pila de operandos*, los operadores buscan los operandos en esta pila, debido a esto tiene notación postfija, es decir los operandos preceden al operador. Además estan la *pila de diccionarios*, la *pila de ejecución*, y la *pila de estado de gráficos*.

Se pueden controlar aspectos del estado de ejecución y tratar los programas como datos. Por medio de operadores se pueden introducir estructuras de control repetitivas y alternativas en el programa de estructura de control secuencial.

Como lenguaje descriptor de páginas, contiene operadores que permiten el emplazamiento de tres tipos de figuras gráficas, son: *texto*, *formas geométricas* y también *imágenes digitalizadas*. Se puede destacar una estructura de datos denominada *estado de gráficos*, en ella se guardan los valores parámetros de control de gráficos, éstos definen el contexto en el cual se ejecutan los operadores gráficos.

El sistema de coordenadas es independiente del dispositivo, y es el propio lenguaje quien se encarga del paso desde su sistema de coordenadas o espacio del usuario, al específico de cada periférico (impresora, filmadora, etc...) y que depende de sus características propias (resolución, orientación de los ejes, etc...). El lenguaje posee varios operadores para el manejo de estado de gráficos, y para cambiar de escala, rotar o transladar el origen del sistema de coordenadas.

El lenguaje también tiene operadores con los que se pueden trazar líneas rectas y curvas con las que se pueden construir figuras o trayectorias (*path*), con cualquier ancho o rellenadas con los tonos deseados. También se puede recortar una zona donde se hacen visibles los gráficos (*clipping path*). Maneja distintos tipos de letra (*font*), también se pueden crear nuevos tipos de letra definidos por el usuario. Las letras son tratadas como dibujos, y se les pueden aplicar operadores gráficos, y se ven afectadas por los cambios en el estado de gráficos. A continuación se muestra un código en lenguaje postscript

```
%
%%BoundingBox: 0 0 576 792
%%TITULO: Ejemplo1.ps
%%Creador: J.M. Cueva Lovelle
%%TAMAÑO: 8*72(576) 11*72(792)
%%Ejemplo de programa Postscript
/inch {72 mul} def
/circulotexto
{  /ycenter exch def /xcenter exch def
   /outlineshow {dup gsave 1 setgray show grestore false charpath stroke} def
   /circuloOviedo
   /fontsize 6 def
   /Times-BoldItalic findfont fontsize scalefont setfont
   {  20 20 340
      { gsave
        /Times-BoldItalic findfont fontsize scalefont setfont
        /fontsize fontsize 17 div fontsize add def
        rotate 0 0 moveto
        (LENGUAJES Y SISTEMAS INFORMATICOS) outlineshow
        grestore
       } for
    } def

   gsave
   xcenter ycenter translate
   0 setlinewidth
   circuloOviedo
   0 0 moveto
   /Times-BoldItalic findfont 15 scalefont setfont
   (LENGUAJES Y SISTEMAS INFORMATICOS) show
   grestore
} def
2 inch 3 inch moveto
currentpoint /yorg exch def /xorg exch def
xorg yorg moveto
gsave
 .5 inch 1.25 inch rmoveto
 /Times-BoldItalic findfont 25 scalefont setfont
 90 rotate (METODOLOGIA DE LA PROGRAMACION II)show
grestore
.2 inch .25 inch rmoveto
/Times-Italic findfont 25 scalefont setfont
(PROCESADORES DE LENGUAJE)show
xorg yorg moveto
1 inch 6 inch rmoveto
/Times-Italic findfont 25 scalefont setfont
(Juan Manuel Cueva Lovelle)show
xorg yorg moveto

1 inch 5.5 inch rmoveto
/Times-Italic findfont 20 scalefont setfont
(TUTORIAS CURSO 98-99)show
xorg yorg moveto
1 inch 5 inch rmoveto
/Times-Italic findfont 12 scalefont setfont
(E.U.I.T.I. de OVIEDO: Viernes 10-12,30 h.)show
xorg yorg moveto
```

```
1 inch 4.5 inch rmoveto
/Times-Italic findfont 12 scalefont setfont
(E.T. Superior I. I. e I. I: Lunes de 17 a 20,30 horas)show
xorg yorg moveto

1.75 inch 2.25 inch rmoveto
currentpoint
circulotexto
showpage
```

En el programa anterior se muestran algunas de las capacidades del lenguaje, siendo el resultado de la ejecución la figura 37, que se ha reducido un 50% para su presentación en este texto. En la primera parte del programa se definen procedimientos que posteriormente son invocados. La unidad de trabajo es 1/72 de pulgada, y con un sistema de coordenadas con origen en el vértice inferior izquierdo del papel se realizan los dibujos y se sitúan los textos. Las líneas que comienzan con el símbolo % son comentarios. Los comentarios de las dos primeras líneas se utilizan por algunos procesadores de texto (por ejemplo *Lotus Manuscript*®) para poder insertar los ficheros postscript como gráficos dentro de sus textos.

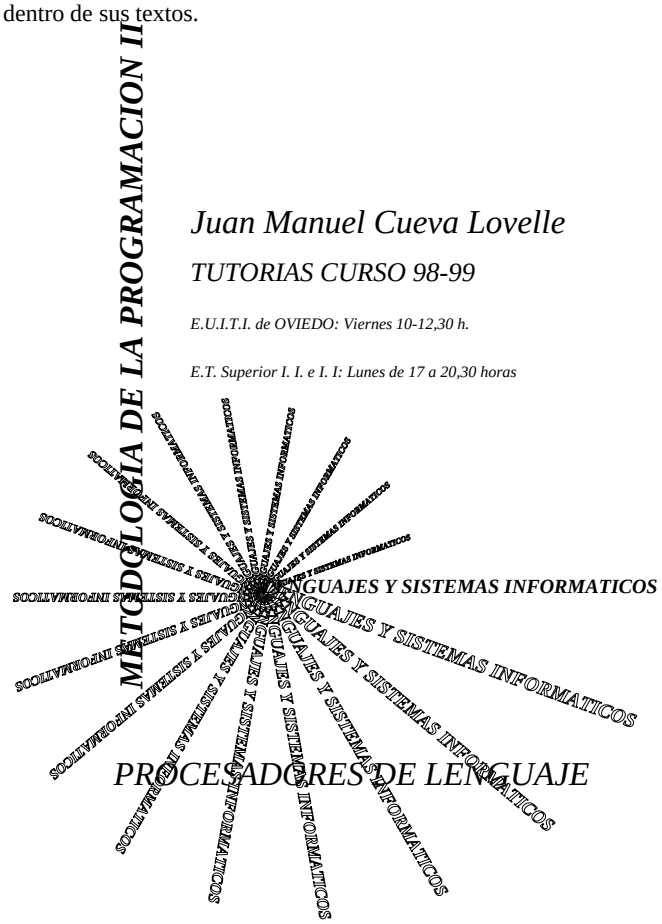


Fig. 41: Resultado del programa Postscript

ANEXO XIII El lenguaje SQL

SQL son las iniciales de *Structured Query Language*, un lenguaje de consulta estructurado de bases de datos creado en la década de los 70 por IBM para trabajar en entornos *mainframe*, y que actualmente está normalizado por el Instituto Americano de Normalización (ANSI). En la actualidad SQL es un lenguaje utilizado tanto en microordenadores como en *mainframes*, y es la base de un gran número de productos comerciales como DB2®, ORACLE®, INGRESS®, INFORMIX®, MULTIBASE®, DBASE®, INTERBASE®,...

El SQL puede trabajar de forma interactiva o en forma inmersa. En la primera forma trabaja como un intérprete se le indican las consultas y emite los informes. El SQL en forma inmersa se utiliza desde dentro de un lenguaje anfitrión: C, COBOL, PL/1, DBASE,... y se compila junto con el código del lenguaje anfitrión.

```
CREATE DATABASE alumnos;
CREATE TABLE matricula (codigo SMALLINT, apellido1 CHAR(20), apellido2 CHAR(20), nombre CHAR(20));
INSERT INTO matricula VALUES (127, "Alonso", "Alonso", "Benjamín");
INSERT INTO matricula VALUES (27, "Pérez", "López", "Margarita");
INSERT INTO matricula VALUES (227, "Fernández", "Alonso", "Luis");
INSERT INTO matricula VALUES (727, "García", "Alonso", "Antonio");
CREATE TABLE parcial1 (codigo SMALLINT, nota DECIMAL(5,2));
INSERT INTO parcial1 VALUES (27, 3.1);
INSERT INTO parcial1 VALUES (127, 8.1);
INSERT INTO parcial1 VALUES (727, 10.0);
INSERT INTO parcial1 VALUES (227, 2.1);
CREATE TABLE parcial2 (codigo SMALLINT, nota DECIMAL(5,2));
INSERT INTO parcial2 VALUES (27, 8.1);
INSERT INTO parcial2 VALUES (127, 8.1);
INSERT INTO parcial2 VALUES (727, 2.0);
INSERT INTO parcial2 VALUES (227, 5.1);
SELECT apellido1, apellido2, nombre, parcial1.nota, parcial2.nota FROM matricula, parcial1, parcial2,
WHERE matricula.codigo=parcial1.codigo AND matricula.codigo=parcial2.codigo ORDER BY apellido1,
apellido2, nombre;
```

Tabla 10: Programa SQL

Las instrucciones o mandatos de SQL pueden dividirse en tres tipos:

- DDL (*Data Definition Language)*. Intrucciones que permiten la definición de datos: creación de bases de datos, diseño de tablas, de índices, etc... Por ejemplo CREATE DATABASE, CREATE TABLE, CREATE INDEX.
- DML (*Data Manipulation Language*). Instrucciones que permiten la manipulación de datos: inserción, modificación y borrado, así como la consulta de los mismos. Por ejemplo INSERT INTO, UPDATE, DELETE FROM, SELECT.
- DCL (*Data Control Language*). Instrucciones que permiten el control de accesos y privilegios en el manejo de datos. Ejemplo GRANT, REVOKE.

En el programa de la tabla 10 se crea una base de datos denominada *alumnos* con tres tablas *matricula*, *parcial1*, y *parcial2*, se introducen algunos datos y se obtiene un listado con los nombres y notas de los alumnos ordenados por orden alfabético.

ANEXO XIV El lenguaje SPSS®

SPSS son las iniciales de *Statistical Package for the Social Sciences*, es decir una aplicación especializada en el tratamiento estadístico de datos diseñada a principios de los años 70. Inicialmente fue preparado para el tratamiento por lotes (*batch*) de datos estadísticos en *mainframes*, para lo cual fue necesario diseñar un lenguaje. Actualmente se puede procesar también interactivamente, y existen versiones para DOS, Windows, UNIX, etc...

En el programa de la tabla 11 se define en la primera instrucción las características de la salida (a disco, ancho 132 columnas, longitud de página 55 líneas, y carácter de salto de página después de cada página). Las instrucciones segunda y tercera definen el título y subtitulo que aparecerá en cada página mientras el programa no encuentre otra instrucción que los modifique. La cuarta instrucción carga los datos a tratar de un fichero de DBASE directamente y asigna los nombres de los campos a las variables a tratar. La quinta instrucción calcula toda la tabla de frecuencias de todas las variables (en este caso todas son cualitativas), con todos sus estadísticos. La sexta instrucción construye todas las tablas de contingencia de todas la variables, determinando varios estadísticos.

```
SET SCREEN=OFF/ DISK=ON/WIDTH=132/LENGTH=55/EJECT=ON.
TITLE 'ESTUDIO ESTADISTICO'.
SUBTITLE 'Ejemplo de un programa en SPSS'.
TRANSLATE FROM 'ESTUDIO.DBF' /TYPE DB3.
FREQUENCIES VARIABLES= ALL /STATISTICS ALL.
CROSSTABS /TABLES= ALL BY ALL /STATISTICS= CHISQ PHI CC /FORMAT= INDEX.
FINISH.
```

Tabla 11: Programa SPSS

ANEXO XV El montador del sistema operativo MS-DOS®

Los compiladores que trabajan en el sistema operativo DOS generan ficheros con la extensión .OBJ, de los distintos módulos y del programa principal, que son encadenados entre sí, y con las bibliotecas por el montador de enlaces (denominado: LINK, TLINK, RTLINK según las distintas marcas comerciales), obteniéndose finalmente un fichero con extensión .EXE que es el fichero objeto completo. Este fichero se carga en memoria por medio del intérprete de comandos COMMAND.COM del sistema operativo DOS, que utiliza la función EXEC del DOS para cargarlo y ejecutarlo.

La función EXEC construye una estructura de datos especial, llamada *prefijo del segmento de programa* (PSP), por encima de la porción residente en memoria de COMMAND.COM, en la zona de programas transitorios. El PSP contiene una serie de conexiones y punteros que necesita el programa de aplicación para ser ejecutado. A continuación, la función EXEC carga el propio programa, justamente encima del PSP, y se realiza cualquier operación de reubicación que sea necesaria. Por último, se cargan los registros dela forma adecuada y se transfiere el control al punto de entrada (*entry point*) del programa. Cuando el programa ha finalizado el trabajo, se llama a una función especial de terminación del DOS, que libera memoria y devuelve el control al programa que hizo que se cargase el programa ejecutado (en general, el propio COMMAND.COM).

Un fichero objeto (.OBJ) es un módulo de un programa, que contiene código y datos, pero también almacena información que puede ser utilizada para combinarlo con otros ficheros objeto para formar un módulo único. El montador de enlaces LINK convierte uno o más ficheros objeto en un fichero ejecutable (.EXE).

Los ficheros .EXE son cargados en memoria por el intérprete de comandos COMMAND.COM, que cambia las direcciones relativas del código máquina a direcciones absolutas, antes de comenzar la ejecución.

Además de los ficheros ejecutables .EXE, el sistema operativo DOS soporta otro tipo de ficheros ejecutables que tienen la extensión .COM. Un fichero .COM no contiene información para el sistema operativo. Existen compiladores y ensambladores que pueden generar directamente un fichero .COM. Cuando el cargador COMMAND.COM arranca un fichero .COM simplemente lo copia del disco a memoria y comienza la ejecución. La carga de los ficheros .COM es mucho más rápida que la de los .EXE, dado que no se realiza ninguna operación. Los ficheros con extensión .COM tienen el código, los datos y la pila (*stack*) en un único segmento. Esto limita el tamaño máximo de los programas .COM a 65024 bytes, 64Kb menos 256 bytes para el PSP (prefijo de segmento de programa) y menos 256 bytes para una pila (*stack*) inicial.

Una aplicación de tipo .COM puede integrarse a partir de distintos módulos objeto separados. Todos los módulos deberán utilizar el mismo segmento nombre de segmento de código y el mismo nombre de clase, y el módulo con el punto de entrada en *offset 0100H* debera ser montado (LINK) en primer lugar. Adicionalmente todos los procedimientos dentro del programa COM deberán disponer de un atributo NEAR, ya que la totalidad del código ejecutable reside en un segmento. Cuando se encadena un programa .COM, el montador de enlaces (LINK) envia el siguiente mensaje: *Warning: no stack segment*, este mensaje puede ser ignorado. La salida del montador (LINK) es un fichero .EXE, que deberá ser convertido en fichero .COM por medio de la utilidad EXE2BIN del DOS, antes de proceder a su ejecución. El fichero .EXE puede borrarse a continuación. También puede crearse con la opción /t del TASM de Borland®.

A diferencia de los programas .COM los programas .EXE sólamente están limitados en cuanto a tamaño por los límites de la memoria disponible en el ordenador. Los programas .EXE suelen también colocar el código, datos y pila en módulos separados en el momento de cargarse.

El cargador del DOS sitúa siempre los programas .EXE en la memoria que se encuentra inmediatamente por encima del PSP, si bien el orden de los segmentos de código, datos y pila puede variar. El fichero .EXE dispone de un encabezamiento, o bloque de información de control, que posee un formato característico, que se puede visualizar con la utilidad EXEHDR del MASM de Microsoft®. El tamaño de este encabezamiento varía según el número de instrucciones de programa que necesitan ser reubicadas durante la carga del mismo, en todo caso siempre es un múltiplo de 512 bytes. Antes de transferir el control al programa, el cargador del DOS se encarga de calcular los valores iniciales del registro de segmento de código (CS) y el registro de puntero a instrucción (IP) a partir de la información del punto de entrada contenida en el encabezamiento del fichero .EXE y de la dirección de carga del programa. Esta información se deduce de una sentencia END situada en uno de los módulos de programa del código fuente. Los registros del segmento de datos (DS) y segmento extra (ES) se disponen de forma que apunten al prefijo del segmento de programa, de forma que el programa pueda acceder al puntero del bloque de entorno, la cola de órdenes y cualquier otra información útil contenida allí. El contenido inicial de los registros de segmento de pila (SS) y puntero de pila (SP) se obtienen del encabezamiento. Esta información se extrae de la declaración de un segmento con el atributo STACK situada en algún lugar del código fuente del programa. Además el espacio reservado para la memoria de la pila puede estar inicializado o no, dependiendo de la definición del segmento de pila; muchos programadores prefieren inicializar la memoria de pila con un conjunto reconocible de datos, de forma que pueda posteriormente volcar a memoria y determinar cuánto espacio de pila se está utilizando realmente en el programa.

La tarea de montaje de un programa .EXE puede realizarse por medio de muchos módulos objeto separados. Cada módulo debe de emplear un único nombre de segmento de código, y sus procedimientos deben llevar el atributo NEAR o bien el atributo FAR, dependiendo de las convenciones de nombres que se utilicen y del tamaño del código ejecutable. El programador deberá cuidar que los módulos enlazados entre sí con el montador (LINK) contengan un sólo segmento con el atributo STACK y un sólo punto de entrada definido con la directriz END del ensamblador. La salida del montador es un fichero con la extensión .EXE, preparado para ser ejecutado directamente.

Desde el punto de vista de la ejecución es interesante indicar que el DOS no se basa en la extensión del fichero cuando desea averiguar el formato de un programa ejecutable. El DOS supone que el programa tiene formato .COM a menos que los dos primeros bytes del fichero sean las letras 'MZ', indicativo de un programa .EXE. La probabilidad de que un programa .COM comience por esta secuencia es mínima, ya que corresponde a las instrucciones DEC BP y POP DX, que no tiene sentido ejecutar hasta que los registros BP y DX contengan algún valor significativo. Además el *checksum* del programa debe coincidir con un valor almacenado en la cabecera; de no ser así, la rutina de carga del DOS deduce que el fichero .EXE está corrupto y muestra un mensaje a tal efecto, interrumpiendo el proceso de ejecución.

El intérprete de comandos COMMAND.COM si usa la extensión del fichero en el caso de encontrar dos programas con el mismo nombre y diferente extensión, cargará siempre el que tiene extensión .COM y no el .EXE. En las versiones anteriores a la 3.30 del MS-DOS ejecutaba el .COM en vez del .EXE incluso en el caso de que el usuario introduzca el nombre del programa completo, con extensión incluida. Hay que señalar que éste es el comportamiento del intérprete de comandos y no del sistema operativo, ya que la rutina de carga de programas del DOS (interrupción 21h, función 4Bh) carga el programa cuyo nombre se ha indicado.

Para más información sobre el sistema operativo DOS pueden consultarse los libros *Microsoft MS-DOS Programmer's Reference* [MICR94] y en castellano *MS-DOS Avanzado* [DUNC86].

Existe un tercer formato ejecutable, denominado *New Executable*, que es utilizado por los programas OS/2 y Windows. Este formato no es más que un .EXE al cual se le ha añadido una segunda cabecera (más extensa), y que contiene un pequeño programa DOS normal, denominado *stub*, que habitualmente muestra un mensaje de aviso y termina. La cabecera "antigua" apunta a este pequeño programa, de forma que si se ejecuta inadvertidamente una aplicación *New* Executable desde DOS, entra en funcionamiento este programa sin que se produzcan mayores problemas. En algunos casos, se sustituye el programa *stub* normal por otro que muestra el *copyright* del fabricante del programa, e instrucciones para la correcta ejecución del mismo. La segunda cabecera contiene los datos reales del programa Windows u OS/2, incluyendo el sistema operativo y versión necesarios para la ejecución, número de módulos, segmentos de código y datos, y varias tablas de segmentos, de recursos (iconos, cursores, *bitmaps*), de referencias externas (llamadas a bibliotecas de enlace dinámico .DLL), de nombres, etc... En el momento de la ejecución Windows u OS/2 ignoran la cabecera antigua y utilizan solamente el código y los datos correspondientes al nuevo formato.

Si se desea conocer en profundidad el formato de los ficheros ejecutables de Windows, puede consultarse el artículo de *Welch* [WELC91]. Sobre Win32 puede consultarse [DUNC93, PIET94].

	Programa .COM	Programa .EXE
Tamaño máximo	65024 bytes	Sin límite
Punto de entrada	PSP:0100H	Definido por la sentencia END
CS a la entrada	PSP	Segmento que contiene el módulo con el punto de entrada
IP a la entrada	0100H	*Ofsset* del punto de entrada dentro de su segmento
DS a la entrada	PSP	PSP
ES a la entrada	PSP	PSP
SS a la entrada	PSP	Segmento con el atributo STACK
SP a la entrada	0FFFEH o palabra superior de la memoria disponible, el que sea menor.	Definido por el segmento con el atributo STACK
Pila a la entrada	Palabra nula	Inicializada o no inicializada
Tamaño de la pila	64Kb menos 256 bytes para PSP y menos el tamaño del código ejecutable	Definido por el segmento con el atributo STACK
Llamada a subrutinas	NEAR	NEAR o FAR
Tamaño del fichero	Tamaño exacto del programa	Tamaño del programa más encabezamiento (múltiplo de 512 bytes)

Tabla 12: Diferencias entre programas .COM y .EXE

ANEXO XVI El lenguaje MUSIM/0

XVI.1 Compilador descendente de cmusim0

a. lexico.h

```
/* Para evitar definiciones múltiples entre módulos */
#ifndef LEXICO_H
#define LEXICO_H
#include <iostream.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
# define TAM_BUFFER 100
class Lexico
 {
      char *nombreFichero;     // Nombre del fichero fuente de entrada
      FILE *entrada;           // Fichero de entrada
      int nl;                  // Número de línea
      int traza;               // Control de traza
      char buffer[TAM_BUFFER]; // Buffer auxiliar de caracteres
      int pBuffer;             // posicion en el buffer auxiliar
```

```
  public:
      Lexico (char *unNombreFichero, int una_traza=0);
      ~Lexico(void);
      char siguienteToken (void);
      void devuelveToken (char token);
      int lineaActual (void) {return nl;};
      int existeTraza(void) {if (traza) return 1; else return 0;}
  };
#endif
```

b. lexico.cpp

```
#include "lexico.h"
  Lexico::Lexico (char *unNombreFichero, int una_traza)
      {
       if((entrada=fopen(unNombreFichero,"r"))==NULL)
         {
          cout<<"No se puede abrir el fichero "<<unNombreFichero<<endl;
          exit (-2);
         }
       if (una_traza) traza=1;
        else traza=0;
        nl=1;      // Se inicializa el contador de líneas
        pBuffer=0; //Se inicializa la posicion del buffer
       }
  Lexico::~Lexico(void)
      {
       fclose(entrada);
      }
  char Lexico::siguienteToken (void)
   {
       char car;
       while ((car=
                ((pBuffer>0) ? buffer[--pBuffer] : getc(entrada))
                )!=EOF)
      { if (car==' ')continue;
        if (car=='\n'){++nl;continue;}
        break;
      }
      if (traza) cout<<"ANALIZADOR LEXICO: Lee el token "<<car<<endl;
      switch (car)
       {
       case 'M':
       case 'R':
       case 'W':           // palabras reservadas
       case '=':           // asignacion
       case '(':           // paréntesis
       case ')':
       case ';':           // separadores
       case '{':
       case '}':
       case '.':           // fin de programa
       case '+':           // operadores aritméticos
       case '-':
       case '*':
       case '/':
       case '%': return (car);
       }
       if (islower(car)) return(car); //variables
        else if (isdigit(car)) return (car); //constantes
        else
            {
             cout<<"ERROR LEXICO: TOKEN DESCONOCIDO"<<endl;
             exit(-4);
            }
       return(car);
   }
```

```
  void Lexico::devuelveToken (char token)
  {
   if (pBuffer>TAM_BUFFER)
     {
     cout<<"ERROR: DESBORDAMIENTO DEL BUFFER DEL ANALIZADOR LEXICO"<<endl;
     exit(-5);
     }
   else
    {
     buffer[pBuffer++]=token;
     if (existeTraza())
      cout<<"ANALIZADOR LEXICO: Recibe en buffer el token "<<token<<endl;
    }
  }
```

c. sintacti.h

```
#ifndef SINTACTI_H
#define SINTACTI_H
#include "lexico.h"
#include "genera.h"
#include <stdlib.h>

class Sintactico
 {
  void programa (void);
  void bloque (void);
  void sentencia (void);
  void otra_sentencia (void);
  void asignacion(void);
  void lectura (void);
  void escritura(void);
  void variable(void);
  void expresion(void);
  void termino(void);
  void mas_terminos(void);
  void factor(void);
  void mas_factores(void);
  void constante(void);
  void errores (int codigo);
  Lexico lexico;
  GeneraCodigo generaCodigo;
  public:
       Sintactico(char *fuente, char *objeto, int traza);
       ~Sintactico(void);
  };
#endif
```

d. sintacti.cpp

```
#include "sintacti.h"

Sintactico::Sintactico(char *fuente, char *objeto, int traza)
        :lexico(fuente, traza), generaCodigo(objeto)
 {
      if (lexico.existeTraza())
             cout<<"INICIO DE ANALISIS SINTACTICO"<<endl;
      programa();
 }
/*****************************************************************/
 Sintactico::~Sintactico(void)
 {
      if (lexico.existeTraza())
        {
          cout<<"FIN DE ANALISIS SINTACTICO"<<endl;
          cout<<"FIN DE COMPILACION"<<endl;
         }
 }
```

```
/*********************************************************************/
void Sintactico::programa(void)
 {
  char token;
  if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <PROGRAMA>"<<endl;
  token=lexico.siguienteToken();
  if (token=='M') generaCodigo.code();
   else errores(8);
  token=lexico.siguienteToken();
  if (token!='{') errores(9);
  bloque();
  token=lexico.siguienteToken();
  if (token=='}')
      {
       generaCodigo.end();
      }
       else errores(2);
 }
/*********************************************************************/
void Sintactico::bloque(void)
 {
  char token;
  if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <BLOQUE>"<<endl;
    sentencia();
    otra_sentencia();
 }
/*********************************************************************/
void Sintactico::otra_sentencia(void)
 {
  char token;
  if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <OTRA_SENTENCIA>"<<endl;
    token=lexico.siguienteToken();
    if (token==';')
     {
      sentencia();
      otra_sentencia();
     }
     else lexico.devuelveToken(token); //vacio
 }
/*********************************************************************/
void Sintactico::sentencia(void)
 {
  char token;
  if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <SENTENCIA>"<<endl;
  token=lexico.siguienteToken();
  if ((token>='a') && (token<='z'))
    {
     lexico.devuelveToken(token);
     asignacion();
    }
  else if (token=='R') lectura();
  else if (token=='W') escritura();
  else errores(6);
 }
/*********************************************************************/
void Sintactico::asignacion()
{
 char token;
  if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <ASIGNACION>"<<endl;
 variable();
 token=lexico.siguienteToken();
```

```
 if (token!='=') errores(3);
 expresion();
 generaCodigo.store();
}
/*************************************************************************/
void Sintactico::variable(void)
{
 char token;
  if (lexico.existeTraza())
              cout<<"ANALISIS SINTACTICO: <VARIABLE>"<<endl;
 token=lexico.siguienteToken();
 if ((token>='a') && (token<='z')) generaCodigo.pusha(token);
  else errores(5);
}
/*************************************************************************/
void Sintactico::expresion(void)
{
 if (lexico.existeTraza())
              cout<<"ANALISIS SINTACTICO: <EXPRESION>"<<endl;
 termino();
 mas_terminos();
}
/*************************************************************************/
void Sintactico::termino(void)
{
  if (lexico.existeTraza())
              cout<<"ANALISIS SINTACTICO: <TERMINO>"<<endl;
 factor();
 mas_factores();
}
/*************************************************************************/
void Sintactico::mas_terminos(void)
{
 char token;
 if (lexico.existeTraza())
              cout<<"ANALISIS SINTACTICO: <MAS_TERMINOS>"<<endl;
 token=lexico.siguienteToken();
 if (token=='+')
  {
      termino();
      generaCodigo.add();
      mas_terminos();
  }
 else if (token =='-')
  {
      termino();
      generaCodigo.neg();
      generaCodigo.add();
      mas_terminos();
  }
  else lexico.devuelveToken(token); // <vacio>
}
/*************************************************************************/
void Sintactico::factor(void)
{
 char token;
 if (lexico.existeTraza())
              cout<<"ANALISIS SINTACTICO: <FACTOR>"<<endl;
 token=lexico.siguienteToken();
 if ((token>='0') && (token<='9'))
                        {
                          lexico.devuelveToken(token);
                          constante();
                         }
 else if (token=='(')
  {
      expresion();
```

```
      token=lexico.siguienteToken();
      if (token!=')') errores(4);
  }
 else
  {
      lexico.devuelveToken(token);
      variable();
      generaCodigo.load();
  }
}
/*********************************************************************/
void Sintactico::mas_factores(void)
{
 char token;
 if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <MAS_FACTORES>"<<endl;
 token=lexico.siguienteToken();
 switch (token)
  {
      case '*':factor();
               generaCodigo.mul();
               mas_factores();
               break;
      case '/':factor();
               generaCodigo.div();
               mas_factores();
               break;
      case '%':factor();
               generaCodigo.mod();
               mas_factores();
               break;
      default: //<vacio>
               lexico.devuelveToken(token);
      }
}
/*********************************************************************/
void Sintactico::lectura(void)
{
 char token;
 token=lexico.siguienteToken();
 if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <LECTURA> "<<token<<endl;
 if((token<'a')||(token>'z')) errores(5);
 generaCodigo.input(token);
}
/*********************************************************************/
 void Sintactico::escritura(void)
{
 char token;
 token=lexico.siguienteToken();
 if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <ESCRITURA> "<<token<<endl;
 if ((token<'a') || (token>'z')) errores(5);
 generaCodigo.output(token);
}
/*********************************************************************/
void Sintactico::constante(void)
{
 char token;
  if (lexico.existeTraza())
             cout<<"ANALISIS SINTACTICO: <CONSTANTE>"<<endl;
 token=lexico.siguienteToken();
 if ((token>='0') && (token<='9')) generaCodigo.pushc(token);
  else errores(7);
}
```

```
/*************************************************************************/
void Sintactico::errores(int codigo)
 {
      cout<<"LINEA "<<lexico.lineaActual();
      cout<<" ERROR SINTACTICO "<<codigo;
      switch (codigo)
       {
        case 1 :cout<<" :ESPERABA UN ;"<<endl;break;
        case 2 :cout<<" :ESPERABA UNA }"<<endl;break;
        case 3 :cout<<" :ESPERABA UN ="<<endl;break;
        case 4 :cout<<" :ESPERABA UN )"<<endl;break;
        case 5 :cout<<" :ESPERABA UN IDENTIFICADOR"<<endl;break;
        case 6 :cout<<" :INSTRUCCION DESCONOCIDA"<<endl;break;
        case 7 :cout<<" :ESPERABA UNA CONSTANTE"<<endl;break;
        case 8 :cout<<" :ESPERABA UNA M DE MAIN"<<endl;break;
        case 9 :cout<<" :ESPERABA UNA {"<<endl;break;
        default:
        cout<<" :NO DOCUMENTADO"<<endl;
       }
      exit(-(codigo+100));
 }
```

e. genera.h

```
#ifndef GENERA_H
#define GENERA_H
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
class GeneraCodigo
 {
  char *nombreFichero; // Nombre del fichero objeto de salida
  FILE *salida;        // Fichero objeto de salida

  public:
      GeneraCodigo(char *unNombreFichero);
      ~GeneraCodigo();
      void code();
      void pushc(char constante);
      void pusha(char direccion);
      void load();
      void store();
      void neg();
      void add();
      void mul();
      void div();
      void mod();
      void input(char direccion);
      void output(char direccion);
      void end();
  };
#endif
```

f. genera.cpp

```
#include "genera.h"
 GeneraCodigo::GeneraCodigo(char *unNombreFichero)
 {
   if ((salida=fopen(unNombreFichero,"w"))==NULL)
      {
      cout<<"No se puede crear el fichero"<<unNombreFichero<<endl;
      exit(-3);
      }
 }
 GeneraCodigo::~GeneraCodigo(void)
 {
  fclose(salida);
 }
```

```
void GeneraCodigo::code()
{
 cout<<".CODE"<<endl;
 fputs(".CODE\n",salida);
}
void GeneraCodigo::pushc(char constante)
{
 cout<<"PUSHC "<<constante<<endl;
 fputs("PUSHC ",salida);
 fputc(constante,salida);
 fputc('\n',salida);
}
void GeneraCodigo::pusha(char direccion)
{
 cout<<"PUSHA "<<direccion<<endl;
 fputs("PUSHA ",salida);
 fputc(direccion,salida);
 fputc('\n',salida);
}
void GeneraCodigo::load()
{
cout<<"LOAD"<<endl;
fputs("LOAD\n",salida);
}
void GeneraCodigo::store()
{
cout<<"STORE"<<endl;
fputs("STORE\n",salida);
}
void GeneraCodigo::neg()
{
cout<<"NEG"<<endl;
fputs("NEG\n",salida);
}
void GeneraCodigo::add()
{
cout<<"ADD"<<endl;
fputs("ADD\n",salida);
}
void GeneraCodigo::mul()
{
cout<<"MUL"<<endl;
fputs("MUL\n",salida);
}
void GeneraCodigo::div()
{
cout<<"DIV"<<endl;
fputs("DIV\n",salida);
}
void GeneraCodigo::mod()
{
cout<<"MOD"<<endl;
fputs("MOD\n",salida);
}

void GeneraCodigo::input(char direccion)
{
 cout<<"INPUT "<<direccion<<endl;
 fputs("INPUT ",salida);
 fputc(direccion,salida);
 fputc('\n',salida);
}
```

```
 void GeneraCodigo::output(char direccion)
 {
  cout<<"OUTPUT "<<direccion<<endl;
  fputs("OUTPUT ",salida);
  fputc(direccion,salida);
  fputc('\n',salida);
 }
 void GeneraCodigo::end()
 {
 cout<<"END"<<endl;
 fputs("END\n",salida);
 }
```

g. cmusim0.cpp

```
/*
  cmusim0.cpp
  Compilador de MUSIM/0 a ENSAMPOCO/0.
  Versión 1.0, en Pascal, Abril 1988.
  Versión 2.0, en C  K&R, Octubre 1990.
  Versión 2.1, en C ANSI, Noviembre 1991.
  Versión 3.0, en C++ , Diciembre 1994.
  Versión 3.1, en C++ , Diciembre 1998.
  Se deben de pasar como argumentos
              - nombre del fichero fuente (MUSIM)
              - nombre del fichero objeto (ENSAMPOCO)
              - opcionalmente se puede mostrar la traza (opción t)
  Es un compilador de un paso, siendo el módulo de análisis sintáctico
  el más importante y el que dirige todo el proceso de traducción.
  El análisis sintáctico es recursivo descendente, siguiendo estrictamente
  la gramática LL(1) del lenguaje MUSIM/0. Obsérvese que existe una
  correspondencia entre los símbolos no terminales de la gramática y los
  métodos utilizados por el analizador sintáctico. La generación de código
  se realiza simultáneamente al análisis.
  (c) Juan Manuel Cueva Lovelle
              Area de Lenguajes y Sistemas Informáticos.
              Dto. de Informática.
              UNIVERSIDAD DE OVIEDO, Diciembre 1998
*/
#include <string.h>
#include <stdlib.h>
#include "sintacti.h"
int main (int argc, char *argv[])
{
 int traza;
 if ((argc < 3) || (argc>4))
  {
      cout<<"Compilador MUSIM/0-EMSAMPOCO/0 Versión 3.0"<<endl;
      cout<<"Forma de uso : CMUSIM Fich_fuente Fich_objeto [t]"<<endl;
      cout<<"         -Fich_fuente (MUSIM/0)"<<endl;
      cout<<"         -Fich_objeto (ENSAMPOCO/0)"<<endl;
      cout<<"         -Si se pone 't' indica que se quiere traza"<<endl<<endl;
      cout<<"         -Si se desea enviar la traza a un fichero escribir:"<<endl;
      cout<<"         CMUSIM0 Fich_fuente Fich_objeto t > fich_traza"<<endl;
      cout<<endl;
      cout<<"(c) Juan Manuel Cueva Lovelle"<<endl;
      cout<<"    Area de Lenguajes y Sistemas Informáticos"<<endl;
      cout<<"    Departamento de Informática"<<endl;
      cout<<"    UNIVERSIDAD DE OVIEDO, Diciembre 1998"<<endl;
      exit(-1);
  }
 if (strcmp(argv[3],"t")) traza=0;
             else traza=1;
 Sintactico sintactico(argv[1], argv[2],traza);
 return 0;
}
```

XVI.2 Traductor de ENSAMPOCO/0 a ensamblador 80x86

a. traducto.h

```
#ifndef TRADUCTO_H
#define TRADUCTO_H
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#define TAM_MAX_OP 7
class Traductor
 {
  FILE *entrada, *salida;
  char opcion;
  char operador[TAM_MAX_OP];
  char direccion;
  void lexico(void);      // Lee del fuente operador y direccion
  void traducir(void);    // Traduce cada tipo de instrucción
  void code(void);        // Un método por cada instrucción */
  void pushc(void);
  void pusha(void);
  void load(void);
  void store(void);
  void neg(void);
  void add(void);
  void mul(void);
  void div(void);
  void mod(void);
  void input(void);
  void output(void);
  void end(void);
  void errores(int);    //Tratamiento de errores
  public:
  Traductor(char *unNombreFicheroEntrada, char *unNombreFicheroSalida);
  ~Traductor(void);
};
#endif
```

b. traducto.cpp

```
#include "traducto.h"
Traductor::Traductor(char *unNombreFicheroEntrada,
                     char *unNombreFicheroSalida)
{
 if ( (entrada= fopen(unNombreFicheroEntrada,"r") ) == NULL)
  {
   cout<<"No se puede abrir el fichero de entrada: ";
   cout<<unNombreFicheroEntrada<<endl;
   exit(-2);
  }
  if ( (salida= fopen(unNombreFicheroSalida,"w") ) == NULL)
  {
   cout<<"No se puede abrir el fichero de salida ";
   cout<<unNombreFicheroSalida<<endl;
   exit(-3);
  }
  cout<<"Traduciendo de ENSAMPOCO/0 a ENSAMBLADOR 80x86 ..."<<endl;
  while (!feof(entrada))
  {
   lexico();
   if (operador[0] != EOF) traducir();
  }
}
/*******************************************************/
```

```
Traductor::~Traductor(void)
{
 if (fcloseall() == EOF)
  {
   cout<<"Error cerrando archivos. Posible pérdida de datos"<<endl;
   exit(-4);
  }
 else cout<<"Final de la traducción"<<endl;
}
/*******************************************************/
void Traductor::lexico(void)
{
// Toma los caracteres del fichero fuente y construye
// el operador y su operando (direccion o constante)

 int  n = 1;   //Contador de caracteres del operador

 if ((operador[0] = fgetc(entrada)) != EOF)
 {
  //Lectura del operador hasta encontrar  blancos o \n
  while (    ((operador[n] = fgetc(entrada)) != ' ')
        && (operador[n] != '\n'))
    n++;
  //Si el operador no ha terminado en \n
  if (operador[n] != '\n')
   {
    //Salta blancos hasta leer la direccion o encontrar \n
    while (   ((direccion = fgetc(entrada)) == ' ')
         && (direccion != '\n'))
      ; //bucle vacío
    //Introduce la marca fin de cadena a operador
    operador[n] = '\0';
    //Si no llego hasta el fin de línea, lee hasta el final
    if (direccion != '\n')
     {
      while ((n = fgetc(entrada)) != '\n')
       ;
     }
    }
  else
     {
      operador[n] = '\0';
     }
  }
}
/*********************************************************************/
void Traductor::traducir(void)
{
 switch (operador[0])
 {
  case '.':if (strcmp (operador,".CODE") == 0) code();
         else errores(1);
         break;
  case 'A':if (strcmp (operador,"ADD") == 0) add();
         else errores(1);
         break;
  case 'D':if (strcmp (operador,"DIV") == 0) div();
         else errores(1);
         break;
  case 'E':if (strcmp (operador,"END") == 0) end();
         else errores(1);
         break;
  case 'I':if (strcmp (operador,"INPUT") == 0) input();
         else errores(1);
         break;
```

```
  case 'L':if (strcmp (operador,"LOAD") == 0) load();
         else errores(1);
         break;
  case 'M':if (strcmp (operador,"MUL") == 0) mul();
         else if (strcmp (operador,"MOD") == 0) mod();
         else errores(1);
         break;
  case 'N':if (strcmp (operador,"NEG") == 0) neg();
         else errores(1);
         break;
  case 'O':if (strcmp (operador,"OUTPUT") == 0) output();
         else errores(1);
         break;
  case 'P':if (strcmp (operador,"PUSHC") == 0) pushc();
         else if (strcmp (operador,"PUSHA") == 0) pusha();
         else errores(1);
         break;
  case 'S':if (strcmp (operador,"STORE") == 0) store();
         else errores(1);
         break;
  default:  errores(1); break;
 }
}
/*************************************************************/
void Traductor::errores(int num)
{
 cout<<"Error de traducción a ensamblador 80x86 "<< num;
 switch (num)
 {
  case 1:cout<<" : Instrucción desconocida"<<endl; break;
  case 2:cout<<" : PUSHA sin dirección"<<endl; break;
  case 3:cout<<" : PUSHC sin constante"<<endl; break;
  case 4:cout<<" : INPUT sin dirección"<<endl; break;
  case 5:cout<<" : OUTPUT sin dirección"<<endl; break;
  default:
       cout<<" : No documentado"<<endl; break;
 }
 exit(-(num+100));
}
/*****************************************************/
void Traductor::pushc(void)
{
 int constante;
 if (direccion == '\n') errores(3);
  else
   {
    constante = direccion -'0';
    fprintf (salida,";\n");
    fprintf (salida,"; PUSHC %d\n",constante);
    fprintf (salida,";\n");
    fprintf (salida,"          XOR  AX,AX\n");
    fprintf (salida,"          MOV  AX,%d\n",constante);
    fprintf (salida,"          PUSH AX\n");
   }
}
/*********************************************************/
void Traductor::pusha(void)
{
 if (direccion == '\n') errores(2);
  else
   {
    direccion= toupper(direccion);
    fprintf (salida,";\n");
    fprintf (salida,"; PUSHA %c\n",direccion);
    fprintf (salida,";\n");
    fprintf (salida,"          XOR  SI,SI\n");
```

```
    fprintf (salida,"          LEA  SI,%c\n",direccion);
    fprintf (salida,"          PUSH SI\n");
   }
}
/*********************************************************************/
void Traductor::load(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";LOAD\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PLOAD\n");
}
/*********************************************************************/
void Traductor::store(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";STORE\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PSTORE\n");
}
/*********************************************************************/
void Traductor::neg(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";NEG\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PNEG\n");
}
/*********************************************************************/
void Traductor::add(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";ADD\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PADD\n");
}
/*********************************************************************/
void Traductor::input(void)
{
if (direccion == '\n') errores(4);
 else
 {
  direccion = toupper(direccion);
  fprintf (salida,";\n");
  fprintf (salida,";INPUT\n");
  fprintf (salida,";\n");
  fprintf (salida,"          CALL PINPUT\n");
  fprintf (salida,"          MOV  %c,DX\n",direccion);
 }
}
/*********************************************************************/
void Traductor::output(void)
{
 if (direccion == '\n') errores(5);
  else
   {
    direccion = toupper(direccion);
    fprintf (salida,";\n");
    fprintf (salida,";OUTPUT\n");
    fprintf (salida,";\n");
    fprintf (salida,"          MOV  AX,%c\n",direccion);
    fprintf (salida,"          CALL POUTPUT\n");
   }
}
```

```
/************************************************************/
void Traductor::mul(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";MUL\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PMUL\n");
}
/************************************************************/
void Traductor::div(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";DIV\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PDIVMOD\n");
 fprintf (salida,"          PUSH AX\n");
}
/************************************************************/
void Traductor::mod(void)
{
 fprintf (salida,";\n");
 fprintf (salida,";MOD\n");
 fprintf (salida,";\n");
 fprintf (salida,"          CALL PDIVMOD\n");
 fprintf (salida,"          PUSH DX\n");
}
/************************************************************/
void Traductor::code(void)
 {
  fprintf(salida,";\n");
  fprintf(salida,"; PROGRAMA ENSAMBLADOR CREADO POR EL TRADUCTOR\n");
  fprintf(salida,";  ENSAMPOCO/0-ENSAMBLADOR 80x86 Versión 3.1\n");
fprintf(salida,";\n");
fprintf(salida,";(c) J.M.Cueva Lovelle\n");
fprintf(salida,";    Area de Lenguajes y Sistemas Informáticos\n");
fprintf(salida,";    Departamento de Informática\n");
fprintf(salida,";    Universidad de Oviedo\n");
fprintf(salida,";\n");
fprintf(salida,"; Macros empleadas:\n");
fprintf(salida,";   ESCRIBE carácter ( visiona un único carácter por pantalla )\n");
fprintf(salida,";   PASALIN          ( pasa de línea con retorno de carro )\n");
fprintf(salida,";   E_CADENA variable_cadena (escribe una cadena)\n");
fprintf(salida,";\n");
fprintf(salida,"ESCRIBE  MACRO CAR\n");
fprintf(salida,"      PUSH AX        ; salvar registros\n");
fprintf(salida,"      PUSH BX\n");
fprintf(salida,"      MOV  AL,CAR\n");
fprintf(salida,"      MOV  BL,0      ; página 0\n");
fprintf(salida,"      MOV  AH,0Eh    ; función: escribir carácter\n");
fprintf(salida,"      INT  10h       ; llamar al BIOS\n");
fprintf(salida,"      POP  BX        ; recuperar registros\n");
fprintf(salida,"      POP  AX\n");
fprintf(salida,"      ENDM\n");
fprintf(salida,"\n");
fprintf(salida,"PASALIN  MACRO\n");
fprintf(salida,"      PUSH AX        ; salvar registros\n");
fprintf(salida,"      PUSH DX\n");
fprintf(salida,"      MOV  AH,2      ; Función de envío de carácter a salida\n");
fprintf(salida,"      MOV  DL,13    ; Envía un carácter de retorno de carro\n");
fprintf(salida,"      INT  21h\n");
fprintf(salida,"      MOV  DL,10    ; Envía un carácter de salto de línea\n");
fprintf(salida,"      INT  21h\n");
fprintf(salida,"      POP  DX        ; recuperar registros\n");
fprintf(salida,"      POP  AX\n");
fprintf(salida,"      ENDM\n");
fprintf(salida,"\n");
fprintf(salida,"E_CADENA MACRO CADENA\n");
```

```
fprintf(salida,"      PUSH AX        ; salvar registros\n");
fprintf(salida,"      PUSH DX\n");
fprintf(salida,"         LEA DX,CADENA \n");
fprintf(salida,"      MOV  AH,9      ; Función que escribe cadena\n");
fprintf(salida,"      INT  21h\n");
fprintf(salida,"      POP  DX        ; recuperar registros\n");
fprintf(salida,"      POP  AX\n");
fprintf(salida,"      ENDM\n");
fprintf(salida,"\n");
fprintf(salida,"; Segmento de pila\n");
fprintf(salida,";\n");
fprintf(salida,"PILA     SEGMENT STACK\n");
fprintf(salida,"         DB      128  DUP (\'PILA\')\n");
fprintf(salida,"PILA     ENDS\n");
fprintf(salida,";\n");
fprintf(salida,"; Segmento de datos\n");
fprintf(salida,";\n");
fprintf(salida,"DATOS    SEGMENT\n");
fprintf(salida,";\n");
fprintf(salida,"; Definición de todas las posibles variables\n");
fprintf(salida,"; que se pueden emplear en MUSIM-ENSAMPOCO\n");
fprintf(salida,";\n");
fprintf(salida,"          A        DW      ?\n");
fprintf(salida,"          B        DW      ?\n");
fprintf(salida,"          C        DW      ?\n");
fprintf(salida,"          D        DW      ?\n");
fprintf(salida,"          E        DW      ?\n");
fprintf(salida,"          F        DW      ?\n");
fprintf(salida,"          G        DW      ?\n");
fprintf(salida,"          H        DW      ?\n");
fprintf(salida,"          I        DW      ?\n");
fprintf(salida,"          J        DW      ?\n");
fprintf(salida,"          K        DW      ?\n");
fprintf(salida,"          L        DW      ?\n");
fprintf(salida,"          M        DW      ?\n");
fprintf(salida,"          N        DW      ?\n");
fprintf(salida,"          O        DW      ?\n");
fprintf(salida,"          P        DW      ?\n");
fprintf(salida,"          Q        DW      ?\n");
fprintf(salida,"          R        DW      ?\n");
fprintf(salida,"          S        DW      ?\n");
fprintf(salida,"          T        DW      ?\n");
fprintf(salida,"          U        DW      ?\n");
fprintf(salida,"          V        DW      ?\n");
fprintf(salida,"          W        DW      ?\n");
fprintf(salida,"          X        DW      ?\n");
fprintf(salida,"          Y        DW      ?\n");
fprintf(salida,"          Z        DW      ?\n");
fprintf(salida,"          _RA       DW      ? ;variables auxiliares\n");
fprintf(salida,"          _INPUT    DB      'ENTRADA>$'\n");
fprintf(salida,"          _OUTPUT   DB      'SALIDA >$'\n");
fprintf(salida,";\n");
fprintf(salida,"DATOS     ENDS\n");
fprintf(salida,";\n");
fprintf(salida,"; Segmento de código\n");
fprintf(salida,";\n");
fprintf(salida,"CODIGO   SEGMENT PARA\n");
fprintf(salida,";\n");
fprintf(salida,"ENTRADA   PROC FAR\n");
fprintf(salida,";\n");
fprintf(salida,"          ASSUME  CS:CODIGO,DS:DATOS,SS:PILA\n");
fprintf(salida,";\n");
fprintf(salida,"          PUSH DS                       ; Inicialización de la pila\n");
fprintf(salida,"          XOR  AX,AX                    ; y carga de direcciones de\n");
fprintf(salida,"          PUSH AX                       ; regreso al DOS.\n");
fprintf(salida,"          MOV  AX,DATOS\n");
fprintf(salida,"          MOV  DS,AX\n");
fprintf(salida,";\n");
fprintf(salida,"; Final de la cabecera del programa (general para todos)\n");
```

```
fprintf(salida,";\n");
fprintf(salida,"; Inicio del cuerpo del programa propiamente dicho\n");
fprintf(salida,";\n");
}
/*************************************************************************/
void Traductor::end(void)
{
fprintf(salida,";\n");
fprintf(salida,"; Saltar los procedimientos.\n");
fprintf(salida,"          JMP  FINPROG\n");
fprintf(salida,";\n");
fprintf(salida,"; PROCEDIMIENTOS.....................................................\n");
fprintf(salida,";\n");
fprintf(salida,"PINPUT    PROC\n");
fprintf(salida,";\n");
fprintf(salida,"; INPUT: lee un número decimal con signo (-9999<=nro<=9999)\n");
fprintf(salida,";     Sólo acepta un - ( o ninguno ) y dígitos.\n");
fprintf(salida,";\n");
fprintf(salida,"          POP _RA\n");
fprintf(salida,"          E_CADENA _INPUT              ; Escribe ENTRADA>\n");
fprintf(salida,"          XOR  DX,DX\n");
fprintf(salida,"          XOR  CX,CX\n");
fprintf(salida,"          MOV  AH,7                       ; Lee una tecla, sin visualizarla\n");
fprintf(salida,"          INT  21h\n");
fprintf(salida,"          CMP  AL,13                   ; ¿Fin lectura? (INTRO)\n");
fprintf(salida,"          JE   FINLEER                 ; Si, salta al final. Asigna 0\n");
fprintf(salida,"          CMP  AL,'-'                  ; ¿Es un menos?\n");
fprintf(salida,"          JNE  POSITIVO                ; No, es un nro. positivo\n");
fprintf(salida,"          MOV  DX,32768                ; Es un nro. negativo. Complemento\n");
fprintf(salida,"          ESCRIBE '-'                  ; Ver el menos\n");
fprintf(salida,"          MOV  AH,7                    ; Lee una tecla, sin visualizarla\n");
fprintf(salida,"          INT  21h\n");
fprintf(salida,"          CMP  AL,13                   ; ¿Fin lectura? (INTRO)\n");
fprintf(salida,"          JE   FINLEER                 ; Si, salta al final\n");
fprintf(salida,"POSITIVO:\n");
fprintf(salida,"          CMP  AL,39h                  ; ¿Es mayor que 9?\n");
fprintf(salida,"          JA   SIGUELEER               ; Si. No se acepta y lee otra tecla\n");
fprintf(salida,"          CMP  AL,30h                  ; ¿Es menor que 0?\n");
fprintf(salida,"          JB   SIGUELEER               ; Si. No se acepta y lee otra tecla\n");
fprintf(salida,"          ESCRIBE AL                   ; Ver el dígito\n");
fprintf(salida,"          XOR  AH,AH                   ; Conversión del nro. en\n");
fprintf(salida,"          SUB  AL,30h                  ; Ascii a dígito decimal\n");
fprintf(salida,"          PUSH AX                     ; Como se acepta, el dígito va a la pila\n");
fprintf(salida,"          INC  CL                      ; Incremento el nro. de cifras.\n");
fprintf(salida,"          CMP  CL,4                    ; ¿Cuatro cifras (9999)?\n");
fprintf(salida,"          JE   NOMASLEER               ; Si. No más cifras, para evitar\n");
fprintf(salida,"                                       ; el posible overflow\n");
fprintf(salida,"SIGUELEER:\n");
fprintf(salida,"          MOV  AH,7                    ; Lee una tecla, sin visualizarla\n");
fprintf(salida,"          INT  21h\n");
fprintf(salida,"          CMP  AL,13                   ; ¿Fin lectura? (INTRO)\n");
fprintf(salida,"          JNE  POSITIVO                ; No, sigue procesando\n");
fprintf(salida,"NOMASLEER:\n");
fprintf(salida,"          CMP  CL,0                    ; ¿Leyó alguna cifra?\n");
fprintf(salida,"          JE   FINLEER                 ; No. Se acabó el proceso\n");
fprintf(salida,"          MOV  AX,1                    ; Peso inicial del dígito\n");
fprintf(salida,"CALCULA:\n");
fprintf(salida,"          POP  BX                      ; Saca de la pila un dígito\n");
fprintf(salida,"          PUSH AX                      ; Guarda en la pila el peso actual\n");
fprintf(salida,"          PUSH DX                      ; Idem con el resultado provisional\n");
fprintf(salida,"          MUL  BX                      ; Multiplica el dígito por su peso\n");
fprintf(salida,"          POP  DX                      ; Recupera el resultado\n");
fprintf(salida,"          ADD  DX,AX                   ; Halla el nuevo resultado parcial\n");
fprintf(salida,"          POP  AX                      ; Recupera el peso actual\n");
fprintf(salida,"          DEC  CX                      ; Decrementa el nro. de cifras\n");
fprintf(salida,"          CMP  CX,0                    ; ¿Es la última cifra?\n");
fprintf(salida,"          JE   FINLEER                 ; Si y se va\n");
fprintf(salida,"          MOV  BX,10                   ; Multiplica por 10 el peso antiguo\n");
```

```
fprintf(salida,"          PUSH DX                       ; para calcular el nuevo peso. Debe\n");
fprintf(salida,"          MUL  BX                       ; meter y luego sacar el resultado\n");
fprintf(salida,"          POP  DX                       ; parcial para no perderlo en el MUL\n");
fprintf(salida,"          JMP  CALCULA                  ; Sigue el bucle\n");
fprintf(salida,"FINLEER:\n");
fprintf(salida,"          CMP  DX,32768                 ;\n");
fprintf(salida,"          JBE  ASIGNA                   ; Cálculo del\n");
fprintf(salida,"          SUB  DX,32769                 ; verdadero\n");
fprintf(salida,"          XOR  AX,AX                    ; complemento\n");
fprintf(salida,"          MOV  AX,DX                    ; del número\n");
fprintf(salida,"          MOV  DX,65535                 ; negativo\n");
fprintf(salida,"          SUB  DX,AX                    ;\n");
fprintf(salida,"ASIGNA:\n");
fprintf(salida,"          PASALIN                       ; Pasa de linea\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PINPUT    ENDP\n");
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"POUTPUT   PROC\n");
fprintf(salida,";\n");
fprintf(salida,"; OUTPUT: escribe número con signo (-32768<=nro<=32768)\n");
fprintf(salida,";\n");
fprintf(salida,"          POP _RA\n");
fprintf(salida,"          E_CADENA _OUTPUT              ; Escribe SALIDA >\n");
fprintf(salida,"          CMP  AX,32768                 ; ¿Es negativo?\n");
fprintf(salida,"          JB   MENOR                    ; No, salta\n");
fprintf(salida,"          ESCRIBE '-'                   ; Ver el menos\n");
fprintf(salida,"          SUB  AX,32768                 ; Calcula el valor\n");
fprintf(salida,"          MOV  DX,AX                    ; absoluto del nro.\n");
fprintf(salida,"          MOV  AX,32768                 ; negativo\n");
fprintf(salida,"          SUB  AX,DX\n");
fprintf(salida,"MENOR:\n");
fprintf(salida,"          MOV  SI,10                    ; Lo divide entre 10 usando SI\n");
fprintf(salida,"          XOR  CX,CX                    ; Cuenta los dígitos que hay en la pila\n");
fprintf(salida,"NO_ES_CERO:\n");
fprintf(salida,"          XOR  DX,DX                    ; Pone la palabra superior N a 0\n");
fprintf(salida,"          DIV  SI                       ; Calcula N/10 y (N mod 10)\n");
fprintf(salida,"          PUSH DX                       ; Pone un dígito en la pila\n");
fprintf(salida,"          INC  CX                       ; Añade un dígito más\n");
fprintf(salida,"          OR   AX,AX                    ; ¿Ya es N = 0?\n");
fprintf(salida,"          JNE  NO_ES_CERO               ; No, continuar\n");
fprintf(salida,"BUCLE_ESCRIBE_DIGITO:\n");
fprintf(salida,"          POP  DX                       ; Pone los dígitos en orden inverso\n");
fprintf(salida,"          ADD  DL,'0'                   ; Convertir en un dígito\n");
fprintf(salida,"          PUSH CX                       ; Guardar el nro. de dígitos\n");
fprintf(salida,"          ESCRIBE DL                    ; Escribir el dígito\n");
fprintf(salida,"          POP  CX                       ; Recuperar nro. dígitos\n");
fprintf(salida,"          LOOP BUCLE_ESCRIBE_DIGITO     ; Obtiene el siguiente dígito\n");
fprintf(salida,"          PASALIN                       ; Pasa de linea\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"POUTPUT   ENDP\n");
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"PADD      PROC\n");
fprintf(salida,";\n");
fprintf(salida,";ADD\n");
fprintf(salida,";\n");
fprintf(salida,"          POP  _RA\n");
fprintf(salida,"          XOR  AX,AX\n");
fprintf(salida,"          XOR  DX,DX\n");
fprintf(salida,"          POP  AX\n");
fprintf(salida,"          POP  DX\n");
fprintf(salida,"          ADD  AX,DX\n");
fprintf(salida,"          PUSH AX\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PADD      ENDP\n");
```

```
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"PNEG      PROC\n");
fprintf(salida,";\n");
fprintf(salida,";NEG\n");
fprintf(salida,";\n");
fprintf(salida,"          POP  _RA\n");
fprintf(salida,"          XOR  AX,AX\n");
fprintf(salida,"          POP  AX\n");
fprintf(salida,"          NEG  AX\n");
fprintf(salida,"          PUSH AX\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PNEG      ENDP\n");
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"PMUL      PROC\n");
fprintf(salida,";\n");
fprintf(salida,"; MUL\n");
fprintf(salida,";\n");
fprintf(salida,"          POP  _RA\n");
fprintf(salida,"          POP  AX\n");
fprintf(salida,"          POP  BX\n");
fprintf(salida,"          IMUL BX\n");
fprintf(salida,"          PUSH AX\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PMUL      ENDP\n");
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"PDIVMOD   PROC\n");
fprintf(salida,"          POP  _RA\n");
fprintf(salida,"          POP  BX\n");
fprintf(salida,"          POP  AX\n");
fprintf(salida,"          CMP  AX,0\n");
fprintf(salida,"          JGE  POSIT\n");
fprintf(salida,"          MOV  DX,0FFFFh\n");
fprintf(salida,"          JMP  DIVIS\n");
fprintf(salida,"POSIT:\n");
fprintf(salida,"          XOR DX,DX\n");
fprintf(salida,"DIVIS:\n");
fprintf(salida,"          IDIV BX\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PDIVMOD   ENDP\n");
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"PLOAD     PROC\n");
fprintf(salida,";\n");
fprintf(salida,";LOAD\n");
fprintf(salida,";\n");
fprintf(salida,"          POP  _RA\n");
fprintf(salida,"          XOR  AX,AX\n");
fprintf(salida,"          XOR  SI,SI\n");
fprintf(salida,"          POP  SI\n");
fprintf(salida,"          MOV  AX,[SI]\n");
fprintf(salida,"          PUSH AX\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PLOAD     ENDP\n");
fprintf(salida,";\n");
fprintf(salida,";\n");
fprintf(salida,"PSTORE    PROC\n");
fprintf(salida,";\n");
fprintf(salida,";STORE\n");
fprintf(salida,";\n");
fprintf(salida,"          POP  _RA\n");
fprintf(salida,"          XOR  AX,AX\n");
fprintf(salida,"          XOR  SI,SI\n");
```

```
fprintf(salida,"          POP  AX\n");
fprintf(salida,"          POP  SI\n");
fprintf(salida,"          MOV  [SI],AX\n");
fprintf(salida,"          PUSH _RA\n");
fprintf(salida,"          RET\n");
fprintf(salida,"PSTORE    ENDP\n");
fprintf(salida,";\n");
fprintf(salida,"FINPROG:\n");
fprintf(salida,";\n");
fprintf(salida,"; Final del cuerpo del programa\n");
fprintf(salida,";\n");
fprintf(salida,"         RET                            ; Retorno al DOS\n");
fprintf(salida,"ENTRADA  ENDP\n");
fprintf(salida,";\n");
fprintf(salida,"CODIGO   ENDS\n");
fprintf(salida,"         END  ENTRADA\n");
}
```

c. tensam0.cpp

```
/*
  tensam0.cpp
  Traductor de ENSAMPOCO/0 a ENSAMBLADOR 80x86
  Versión 1.0, en C K&R,  Octubre 1989.
  Versión 2.0, en C K&R,  Octubre 1990.
  Versión 2.1, en C ANSI, Noviembre 1991.
  Versión 2.2, en C ANSI, Octubre 1992.
  Versión 3.0, en C++   , Diciembre 1994.
  Versión 3.1, en C++   , Diciembre 1998.
  Se deben de pasar como argumentos
             - nombre del fichero ENSAMPOCO/0
             - nombre del fichero ENSAMBLADOR
  La traducción se realiza instrucción a instrucción. Puede observarse
  como se ha construido un método para cada instrucción de Ensampoco/0,
  dicho método realiza la traducción. En el método code se escribe la
  cabecera del programa ensamblador, y en el método end se escriben los
  procedimientos construidos en ensamblador.
  El programa va explorando el fichero de entrada con el método lexico, y
  traduciendo con el metodo traducir.
  (c) Juan Manuel Cueva Lovelle
      Area de Lenguajes y Sistemas Informáticos
      Departamento de Informática
      UNIVERSIDAD DE OVIEDO, Diciembre 1998
*/

#include "traducto.h"
int main (int argc,char *argv[])
{
 if ((argc < 3) || (argc>3))
  {
   cout<<"Traductor EMSAMPOCO/0 - ENSAMBLADOR 80x86 Versión 3.1"<<endl;
   cout<<"Forma de uso : TENSAM Fich_poc Fich_asm"<<endl;
   cout<<endl;
   cout<<"     -Fich_poc fichero Ensampoco"<<endl;
   cout<<"     -Fich_asm fichero Ensamblador"<<endl<<endl;
   cout<<"(c) Juan Manuel Cueva Lovelle"<<endl;
   cout<<"    Area de Lenguajes y Sistemas Informáticos"<<endl;
   cout<<"    Departamento de Informática"<<endl;
   cout<<"    UNIVERSIDAD DE OVIEDO, Diciembre 1998"<<endl;
   exit(-1);
  }
 Traductor traductor(argv[1], argv[2]);
 return 0;
}
```

XVI.3 Intérprete de ENSAMPOCO/0

a. interpre.h

```
#ifndef INTERPRE_H
#define INTERPRE_H
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream.h>
#define TAM_MAX_INS 7
struct pila {
           char valor;
           struct pila *sucesor;
           };
typedef struct pila ELEMENTO;
typedef ELEMENTO *PUNTERO;
class Interprete
 {
  FILE *entrada;                 //Descriptor del fichero de entrada
  char instruccion[TAM_MAX_INS];
  char parametro;
  PUNTERO tope;       //puntero al tope de la pila
  int memoria[26];    //memoria: array de 'a'..'z'de int
  void lexico(void);    // Coge las instrucciones del fichero fuente
  void evaluador(void); // Evalúa las expresiones
  void error(int);      // Manejo de errores
  void code(void);      // Instrucciones de ENSAMPOCO/0
  void pusha(void);
  void pushc(void);
  void load(void);
  void store(void);
  void neg(void);
  void add(void);
  void mul(void);
  void div(void);
  void mod(void);
  void input(void);
  void output(void);
  void end (void);
  public:
  Interprete (char *unNombreFicheroEntrada);
  ~Interprete (void);
 };
#endif
```

b. interpre.cpp

```
#include "interpre.h"
Interprete::Interprete (char *unNombreFicheroEntrada)
{
 int i;
 if ((entrada = fopen(unNombreFicheroEntrada,"r")) == NULL ) error(0);
 for(i = 0; i<26; i++)  memoria[i] = 0;  //inicialización a cero
 while (!feof(entrada))
    {
    lexico();
    if (instruccion[0] != EOF) evaluador();
    }
}
/*************************************************************************/
Interprete::~Interprete (void)
{
 fcloseall();
}
/*************************************************************************/
```

```
void Interprete::lexico(void)
{
int  n;
if ((instruccion[0] = fgetc(entrada)) != EOF)
 {
 n = 1;
 while (((instruccion[n] = fgetc(entrada)) != ' ') && (instruccion[n] != '\n'))
  n++;
 if (instruccion[n] != '\n')
  {
  while (((parametro = fgetc(entrada)) == ' ') && (parametro != '\n'))
  ;
  instruccion[n] = '\0';
  if (parametro != '\n')
   {
   while ((n = fgetc(entrada)) != '\n')
   ;
   }
  }
 else
  {
  instruccion[n] = '\0';
  }
 }
}
/*********************************************/
void Interprete::evaluador(void)
{
 switch (instruccion[0])
 {
   case '.': if (strcmp (instruccion,".CODE") == 0) code();
             else error(1);
             break;
   case 'A': if (strcmp (instruccion,"ADD") == 0) add();
             else error(1);
             break;
   case 'D': if (strcmp (instruccion,"DIV") == 0) div();
             else error(1);
             break;
   case 'E': if (strcmp (instruccion,"END") == 0) end();
             else error(1);
             break;
   case 'I': if (strcmp (instruccion,"INPUT") == 0) input();
             else error(1);
             break;
   case 'L': if (strcmp (instruccion,"LOAD") == 0) load();
             else error(1);
             break;
   case 'M': if (strcmp (instruccion,"MUL") == 0) mul();
             else if (strcmp (instruccion,"MOD") == 0) mod();
             else error(1);
             break;
   case 'N': if (strcmp (instruccion,"NEG") == 0) neg();
             else error(1);
             break;
   case 'O': if (strcmp (instruccion,"OUTPUT") == 0) output();
             else error(1);
             break;
   case 'P': if (strcmp (instruccion,"PUSHC") == 0) pushc();
             else if (strcmp (instruccion,"PUSHA") == 0)
             pusha();
             else error(1);
             break;
```

```
   case 'S': if (strcmp (instruccion,"STORE") == 0) store();
           else error(1);
           break;
   default : error(1); break;
 }
}
/*************************************************************/
void Interprete::error(int num)
// Manejo de errores
{
 cout<<"Error "<<num;
 switch (num)
  {
   case 0: cout<<": No encuentra el fichero fuente"<<endl; break;
   case 1: cout<<": Instrucción desconocida"<<endl;break;
   case 2: cout<<": PUSHA sin variable"<<endl;break;
   case 3: cout<<": PUSHC sin constante"<<endl;break;
   case 4: cout<<": INPUT sin variable"<<endl;break;
   case 5: cout<<": OUTPUT sin vaariable"<<endl;break;
   default:
         cout<<": No documentado"<<endl; break;
 }
}
/*********************************************************************/
void Interprete::input(void)
{
 int numero;
 cout<<"Entrada "<< parametro << " > ";
 cin>>numero;
 memoria[parametro-'a'] = numero;
 }
/*********************************************************************/
void Interprete::output(void)
{
 cout<<"Salida "<<parametro<<" > "<<memoria[parametro-'a']<<endl;
}
/*********************************************************************/
void Interprete::pushc(void)
{
 PUNTERO p;
 p = (PUNTERO) malloc (sizeof(ELEMENTO));
 p->valor = parametro-'0';
 p->sucesor = tope;
 tope = p;
}
/*********************************************************************/
void Interprete::pusha(void)
{
 PUNTERO p;
 p = (PUNTERO) malloc (sizeof(ELEMENTO));
 p->valor = parametro;
 p->sucesor = tope;
 tope = p;
}
/*********************************************************************/
void Interprete::load(void)
{
 tope->valor = memoria[tope->valor-'a'];
}
/*********************************************************************/
void Interprete::store(void)
{
 PUNTERO p,q;
 p = tope;
 q = tope->sucesor;
```

```
 if (q->sucesor!=NULL) tope = q->sucesor;
   else tope=NULL;
 memoria[(q->valor)-'a'] = p->valor;
 free((PUNTERO)p);
 free((PUNTERO)q);
}
/*********************************************************************/
void Interprete::add(void)
{
  PUNTERO p;
  p = tope;
  tope = tope->sucesor;
  tope->valor = tope->valor+p->valor;
  free((PUNTERO)p);
 }
/*********************************************************************/
void Interprete::neg(void)
{
 tope->valor = -tope->valor;
}
/*********************************************************************/
void Interprete::mul(void)
{
 PUNTERO p;
 p = tope;
 tope = tope->sucesor;
 tope->valor = tope->valor*p->valor;
 free((PUNTERO)p);
}
/*********************************************************************/
void Interprete::div(void)
{
 PUNTERO p;
 p = tope;
 tope = tope->sucesor;
 tope->valor = (int) (tope->valor / p->valor);
 free((PUNTERO)p);
}
/*********************************************************************/
void Interprete::mod(void)
{
  PUNTERO p;
  p = tope;
  tope = tope->sucesor;
  tope->valor = (( (int)tope->valor) % ( (int) p->valor) );
  free((PUNTERO)p);
 }
/*********************************************************************/
void Interprete::code(void) { }
/*********************************************************************/
void Interprete::end(void) { }
```

c. iensam0.cpp

```
/*
   iensan0.cpp
   Intérprete de ENSAMPOCO/0
   Versión 1.0, en Pascal, Abril 1988.
   Versión 2.0, en C K&R, Octubre 1990.
   Versión 2.1, en C ANSI, Noviembre 1991.
   Versión 3.0, en C++, Diciembre 1994.
   Versión 3.1, en C++, Diciembre 1998.
```

```
   (c) Juan Manuel Cueva Lovelle
       Area de Lenguajes y Sistemas Informáticos.
       Departamento de Informática.
       Universidad de Oviedo, Diciembre 1998.
*/
#include "interpre.h"
int main(int argc, char **argv)
{
 if (argc != 2)
  {
   cout<<endl;
   cout<<"IENSAM0 : Intérprete de ENSAMPOCO/0. Versión 3.1"<<endl;
   cout<<endl;
   cout<<"      (c) Juan Manuel Cueva Lovelle"<<endl;
   cout<<"          Area de Lenguajes y Sistemas Informáticos"<<endl;
   cout<<"          Departamento de Informática"<<endl;
   cout<<"          Universidad de Oviedo, Diciembre 1998"<<endl;
   cout<<endl;
   cout<<"Forma de uso:"<<endl;
   cout<<"               iensam fich_poc"<<endl;
   cout<<endl;
   cout<<"donde fich_poc es el fichero de ENSAMPOCO/0"<<endl;
   exit(-1);
  }
 Interprete interprete (argv[1]);
 return 0;
}
```

XVI.4 compila.bat

```
cmusim %1.mus %1.poc
tensam %1.poc %1.asm
masm %1,,,,
rem -- también puede hacerse con TASM
link %1,,,,
rem -- también puede hacerse con TLINK
```

XVI.5 interpre.bat

```
cmusim %1.mus %1.poc
iensam %1.poc
```

XVI.6 Ejem1.mus

```
M
{
 R a;
 R b;
 R c;
 R d;
 z = ( a + b ) * ( c + d );
 W z
}
```

XVI.7 Ejem1.poc

```
.CODE
INPUT a
INPUT b
INPUT c
INPUT d
PUSHA z
PUSHA a
LOAD
PUSHA b
LOAD
ADD
PUSHA c
LOAD
PUSHA d
```

```
                LOAD
                ADD
                MUL
                STORE
                OUTPUT z
                END
```

XVI.8 Ejem1.asm

```
;
; PROGRAMA ENSAMBLADOR CREADO POR EL TRADUCTOR
;  ENSAMPOCO/0-ENSAMBLADOR 80x86 Versión 3.1
;
;(c) J.M.Cueva Lovelle
;    Area de Lenguajes y Sistemas Informáticos
;    Departamento de Informática
;    Universidad de Oviedo
;
; Macros empleadas:
;   ESCRIBE carácter ( visiona un único carácter por pantalla )
;   PASALIN          ( pasa de línea con retorno de carro )
;   E_CADENA variable_cadena (escribe una cadena)
;
ESCRIBE  MACRO CAR
       PUSH AX        ; salvar registros
       PUSH BX
       MOV  AL,CAR
       MOV  BL,0      ; página 0
       MOV  AH,0Eh    ; función: escribir carácter
       INT  10h       ; llamar al BIOS
       POP  BX        ; recuperar registros
       POP  AX
       ENDM
PASALIN  MACRO
       PUSH AX        ; salvar registros
       PUSH DX
       MOV  AH,2      ; Función de envío de carácter a salida
       MOV  DL,13    ; Envía un carácter de retorno de carro
       INT  21h
       MOV  DL,10    ; Envía un carácter de salto de línea
       INT  21h
       POP  DX        ; recuperar registros
       POP  AX
       ENDM
E_CADENA MACRO CADENA
       PUSH AX        ; salvar registros
       PUSH DX
         LEA DX,CADENA
       MOV  AH,9      ; Función que escribe cadena
       INT  21h
       POP  DX        ; recuperar registros
       POP  AX
       ENDM
; Segmento de pila
;
PILA     SEGMENT STACK
         DB      128  DUP ('PILA')
PILA     ENDS
;
; Segmento de datos
;
DATOS    SEGMENT
;
; Definición de todas las posibles variables
; que se pueden emplear en MUSIM-ENSAMPOCO
;
          A        DW      ?
          B        DW      ?
```

```
          C         DW      ?
          D         DW      ?
          E         DW      ?
          F         DW      ?
          G         DW      ?
          H         DW      ?
          I         DW      ?
          J         DW      ?
          K         DW      ?
          L         DW      ?
          M         DW      ?
          N         DW      ?
          O         DW      ?
          P         DW      ?
          Q         DW      ?
          R         DW      ?
          S         DW      ?
          T         DW      ?
          U         DW      ?
          V         DW      ?
          W         DW      ?
          X         DW      ?
          Y         DW      ?
          Z         DW      ?
          _RA       DW      ? ;variables auxiliares
          _INPUT    DB      'ENTRADA>$'
          _OUTPUT   DB      'SALIDA >$'
;
DATOS     ENDS
;
; Segmento de código
;
CODIGO   SEGMENT PARA
;
ENTRADA   PROC FAR
;
          ASSUME  CS:CODIGO,DS:DATOS,SS:PILA
;
          PUSH DS                       ; Inicialización de la pila
          XOR  AX,AX                    ; y carga de direcciones de
          PUSH AX                       ; regreso al DOS.
          MOV  AX,DATOS
          MOV  DS,AX
;
; Final de la cabecera del programa (general para todos)
;
; Inicio del cuerpo del programa propiamente dicho
;
;
;INPUT
;
          CALL PINPUT
          MOV  A,DX
;
;INPUT
;
          CALL PINPUT
          MOV  B,DX
;
;INPUT
;
          CALL PINPUT
          MOV  C,DX
;
;INPUT
;
          CALL PINPUT
          MOV  D,DX
;
```

```
; PUSHA Z
;
          XOR  SI,SI
          LEA  SI,Z
          PUSH SI
;
; PUSHA A
;
          XOR  SI,SI
          LEA  SI,A
          PUSH SI
;
;LOAD
;
          CALL PLOAD
;
; PUSHA B
;
          XOR  SI,SI
          LEA  SI,B
          PUSH SI
;
;LOAD
;
          CALL PLOAD
;
;ADD
;
          CALL PADD
;
; PUSHA C
;
          XOR  SI,SI
          LEA  SI,C
          PUSH SI
;
;LOAD
;
          CALL PLOAD
;
; PUSHA D
;
          XOR  SI,SI
          LEA  SI,D
          PUSH SI
;
;LOAD
;
          CALL PLOAD
;
;ADD
;
          CALL PADD
;
;MUL
;
          CALL PMUL
;
;STORE
;
          CALL PSTORE
;
;OUTPUT
;
          MOV  AX,Z
          CALL POUTPUT
;
; Saltar los procedimientos.
          JMP  FINPROG
```

```
;
; PROCEDIMIENTOS.....................................................
;
PINPUT    PROC
;
; INPUT: lee un número decimal con signo (-9999<=nro<=9999)
;      Sólo acepta un - ( o ninguno ) y dígitos.
;
          POP _RA
          E_CADENA _INPUT              ; Escribe ENTRADA>
          XOR  DX,DX
          XOR  CX,CX
          MOV  AH,7                 ; Lee una tecla, sin visualizarla
          INT  21h
          CMP  AL,13                   ; ¿Fin lectura? (INTRO)
          JE   FINLEER                 ; Si, salta al final. Asigna 0
          CMP  AL,'-'                  ; ¿Es un menos?
          JNE  POSITIVO                ; No, es un nro. positivo
          MOV  DX,32768                ; Es un nro. negativo. Complemento
          ESCRIBE '-'                  ; Ver el menos
          MOV  AH,7                    ; Lee una tecla, sin visualizarla
          INT  21h
          CMP  AL,13                   ; ¿Fin lectura? (INTRO)
          JE   FINLEER                 ; Si, salta al final
POSITIVO:
          CMP  AL,39h                  ; ¿Es mayor que 9?
          JA   SIGUELEER               ; Si. No se acepta y lee otra tecla
          CMP  AL,30h                  ; ¿Es menor que 0?
          JB   SIGUELEER               ; Si. No se acepta y lee otra tecla
          ESCRIBE AL                   ; Ver el dígito
          XOR  AH,AH                   ; Conversión del nro. en
          SUB  AL,30h                  ; Ascii a dígito decimal
          PUSH AX                      ; Como se acepta, el dígito va a la pila
          INC  CL                      ; Incremento el nro. de cifras.
          CMP  CL,4                    ; ¿Cuatro cifras (9999)?
          JE   NOMASLEER               ; Si. No más cifras, para evitar
                                       ; el posible overflow
SIGUELEER:
          MOV  AH,7                    ; Lee una tecla, sin visualizarla
          INT  21h
          CMP  AL,13                   ; ¿Fin lectura? (INTRO)
          JNE  POSITIVO                ; No, sigue procesando
NOMASLEER:
          CMP  CL,0                    ; ¿Leyó alguna cifra?
          JE   FINLEER                 ; No. Se acabó el proceso
          MOV  AX,1                    ; Peso inicial del dígito
CALCULA:
          POP  BX                      ; Saca de la pila un dígito
          PUSH AX                      ; Guarda en la pila el peso actual
          PUSH DX                      ; Idem con el resultado provisional
          MUL  BX                      ; Multiplica el dígito por su peso
          POP  DX                      ; Recupera el resultado
          ADD  DX,AX                   ; Halla el nuevo resultado parcial
          POP  AX                      ; Recupera el peso actual
          DEC  CX                      ; Decrementa el nro. de cifras
          CMP  CX,0                    ; ¿Es la última cifra?
          JE   FINLEER                 ; Si y se va
          MOV  BX,10                   ; Multiplica por 10 el peso antiguo
          PUSH DX                      ; para calcular el nuevo peso. Debe
          MUL  BX                      ; meter y luego sacar el resultado
          POP  DX                      ; parcial para no perderlo en el MUL
          JMP  CALCULA                 ; Sigue el bucle
FINLEER:
          CMP  DX,32768                ;
          JBE  ASIGNA                  ; Cálculo del
          SUB  DX,32769                ; verdadero
          XOR  AX,AX                   ; complemento
          MOV  AX,DX                   ; del número
          MOV  DX,65535                ; negativo
```

```
          SUB  DX,AX                    ;
ASIGNA:
          PASALIN                       ; Pasa de linea
          PUSH _RA
          RET
PINPUT    ENDP
;
;
POUTPUT   PROC
;
; OUTPUT: escribe número con signo (-32768<=nro<=32768)
;
          POP _RA
          E_CADENA _OUTPUT              ; Escribe SALIDA >
          CMP  AX,32768                 ; ¿Es negativo?
          JB   MENOR                    ; No, salta
          ESCRIBE '-'                   ; Ver el menos
          SUB  AX,32768                 ; Calcula el valor
          MOV  DX,AX                    ; absoluto del nro.
          MOV  AX,32768                 ; negativo
          SUB  AX,DX
MENOR:
          MOV  SI,10                    ; Lo divide entre 10 usando SI
          XOR  CX,CX                    ; Cuenta los dígitos que hay en la pila
NO_ES_CERO:
          XOR  DX,DX                    ; Pone la palabra superior N a 0
          DIV  SI                       ; Calcula N/10 y (N mod 10)
          PUSH DX                       ; Pone un dígito en la pila
          INC  CX                       ; Añade un dígito más
          OR   AX,AX                    ; ¿Ya es N = 0?
          JNE  NO_ES_CERO               ; No, continuar
BUCLE_ESCRIBE_DIGITO:
          POP  DX                       ; Pone los dígitos en orden inverso
          ADD  DL,'0'                   ; Convertir en un dígito
          PUSH CX                       ; Guardar el nro. de dígitos
          ESCRIBE DL                    ; Escribir el dígito
          POP  CX                       ; Recuperar nro. dígitos
          LOOP BUCLE_ESCRIBE_DIGITO     ; Obtiene el siguiente dígito
          PASALIN                       ; Pasa de linea
          PUSH _RA
          RET
POUTPUT   ENDP
;
;
PADD      PROC
;
;ADD
;
          POP  _RA
          XOR  AX,AX
          XOR  DX,DX
          POP  AX
          POP  DX
          ADD  AX,DX
          PUSH AX
          PUSH _RA
          RET
PADD      ENDP
;
;
PNEG      PROC
;
;NEG
;
          POP  _RA
          XOR  AX,AX
          POP  AX
          NEG  AX
          PUSH AX
```

```
          PUSH _RA
          RET
PNEG      ENDP
;
;
PMUL      PROC
;
; MUL
;
          POP  _RA
          POP  AX
          POP  BX
          IMUL BX
          PUSH AX
          PUSH _RA
          RET
PMUL      ENDP
;
;
PDIVMOD   PROC
          POP  _RA
          POP  BX
          POP  AX
          CMP  AX,0
          JGE  POSIT
          MOV  DX,0FFFFh
          JMP  DIVIS
POSIT:
          XOR DX,DX
DIVIS:
          IDIV BX
          PUSH _RA
          RET
PDIVMOD   ENDP
;
;
PLOAD     PROC
;
;LOAD
;
          POP  _RA
          XOR  AX,AX
          XOR  SI,SI
          POP  SI
          MOV  AX,[SI]
          PUSH AX
          PUSH _RA
          RET
PLOAD     ENDP
;
;
PSTORE    PROC
;
;STORE
;
          POP  _RA
          XOR  AX,AX
          XOR  SI,SI
          POP  AX
          POP  SI
          MOV  [SI],AX
          PUSH _RA
          RET
PSTORE    ENDP
;
FINPROG:
;
; Final del cuerpo del programa
;
```

```
        RET                             ; Retorno al DOS
ENTRADA ENDP
;
CODIGO  ENDS
        END  ENTRADA
```

ANEXO XVII Intérprete PIPO

a. rpn.h

```
#ifndef RPN_H
#define RPN_H
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#define MAX_INS 20        // Tamaño máximo de una instrucción
#define NUMERO '0'        // Atributo de la instrucción número
#define GRANDE '9'        // La instrucción es demasiado grande
#define TAM_BUFFER 100    // Tamaño del buffer del analizador léxico

class Rpn
{
 double *tope;              // Señala al tope de la pila
 double *inicio;            // Señala el principio de la pila
 double *fin;               // Señala el final de la pila
 int tamPila;               // Tamaño de la pila
 char instruccion[MAX_INS]; // Instrucción
 int atributo_ins;          // Atributo de la instrucción
 char buffer[TAM_BUFFER];   // Buffer del analizador de instrucciones
 int pBuffer;               // Posición en el buffer
 void evaluador(void);       // Evaluador de instrucciones
 void prompt(void);          // Prompt del intérprete
 void ayuda (void);          // Información de ayuda
 void errorFatal (int cod);  // Errores que implican finalización
 void avisoError (int cod);  // Errores que dan un mensaje y continuan
 int toma_instruccion(void); // Toma la siguiente instrucción
 int toma_char(void);        // Coge un carácter para formar la instrucción
 void devuelve_char(int c);  // Devuelve un carácter al buffer
 void borra(void);           // Borra el contenido de la pila
 double push(double);        // Empuja un valor en la pila
 double pop(void);           // Saca un valor de la pila
  public:
  Rpn(int tam_pila);
 ~Rpn(void);
};
#endif
```

b. rpn.cpp

```
#include "rpn.h"
Rpn::Rpn(int tamPila)
{
 tope = (double *) malloc (tamPila*sizeof(double));
 if ( tope == NULL) errorFatal(1);
 pBuffer=0;
 inicio=tope;
 fin=tope+tamPila-1;
 ayuda();
 while ((atributo_ins=toma_instruccion()) != EOF) evaluador();
}
Rpn::~Rpn(void)
{
 free(tope);
}
```

```
void Rpn::prompt(void)
{
 cout<<"PIPO> ";
}
void Rpn::errorFatal(int cod)
// Después del error finaliza el programa
{
 cout<<"Error fatal "<<cod;
 switch (cod)
  {
   case 1: cout<<": No hay memoria disponible para crear la pila"<<endl;
         break;
   case 2: cout<<": Pila completa. Se trató de introducir un valor"<<endl;
         break;
   case 3: cout<<": Pila vacia. Se trató de sacar un valor"<<endl;
         break;
   case 4: cout<<": Buffer lleno. No se puede devolver carácter"<<endl;
         break;
   default:
         cout<<": No documentado"<<endl;
   }
 exit (-cod);
}
void Rpn::avisoError(int cod)
// Permite continuar después del error
{
 cout<<"Aviso de error "<<cod;
 switch (cod)
  {
   case 1: cout<<": División por cero"<<endl;break;
   case 2: cout<<": El operando introducido es demasiado largo -> ";break;
   case 3: cout<<": El operador introducido no es válido -> ";break;
   default:
         cout<<": No documentado"<<endl;break;
   }
}
void Rpn::ayuda(void)
{
 cout<<"PIPO: Intérprete de una calculadora en notación polaca inversa"<<endl;
 cout<<endl;
 cout<<"(c) Versión 3.0"<<endl;
 cout<<"    Juan Manuel Cueva Lovelle"<<endl;
 cout<<"    Area de Lenguajes y Sistemas Informáticos"<<endl;
 cout<<"    Universidad de Oviedo, Diciembre 1994"<<endl;
 cout<<endl;
 cout<<"    Utiliza una pila LIFO para realizar la evaluación de"<<endl;
 cout<<"    las expresiones. PIPO es abreviatura de PIla POlaca"<<endl;
 cout<<"    Introduzca una expresión en notación postfija o polaca"<<endl;
 cout<<"    inversa, es decir los operandos en primer lugar y luego"<<endl;
 cout<<"    el operador, Para evaluarla pulse  =  y después <intro>"<<endl;
 cout<<endl;
 cout<<"    Operadores disponibles: +, -, *, y /. Operan sobre reales"<<endl;
 cout<<"    Otras operaciones: c (inicializa la pila), f (finaliza)"<<endl;
 cout<<"                       h (ayuda), p (visualiza la pila)"<<endl;
 cout<<endl;
 cout<<"    Para salir: f y después <intro>."<<endl;
 prompt();
}
void Rpn::evaluador(void)
{
 double op2; // auxiliares
 double *p;
   switch(atributo_ins)
   {
    case NUMERO:
              push(atof(instruccion));
              break;
    case '+':
```

```
            push(pop()+pop());
            break;
    case '*':
            push( pop() * pop() );
            break;
    case '-':                   /* No es commutativo */
            op2=pop();
            push(pop() - op2);
            break;
    case '/':                   /* No es commutativo */
            op2=pop();
            if (op2 != 0.0)  push(pop()/op2);
              else  avisoError(1);
            break;
     case '=':
            // muestra el contenido del tope de la pila
            cout<< "Resultado> "<<setprecision(6)<<push(pop())<< endl;
            prompt();
            break;
     case 'c':
     case 'C':
            //reinicializa la pila
            borra();
            cout<<"Pila eliminada"<<endl;
            prompt();
            break;
     case 'h':
     case 'H': ayuda(); break;
     case 'p':
     case 'P': // Visualiza los contenidos actuales en la pila
            cout<<"Estado actual de la pila"<<endl;
            if (tope<inicio) { avisoError(3); break;}
            if (tope==inicio) {cout<<"Vacia"<<endl;prompt();break;}
            p=inicio;
            do
             {
              cout<<*p<<endl;
              p++;
             }
            while (p<tope);
            prompt();
            break;
     case 'f':                /* Finalización */
     case 'F':  cout<<"Fin"<<endl;
            exit(0);
     case GRANDE: avisoError(2);
              cout<<instruccion<<endl;
              break;
     default:
            avisoError(3);
            cout<<atributo_ins<<endl;
            break;
    }
}
double Rpn::push(double f)    /* Introduce el valor f en la pila */
{
 if (tope>fin) errorFatal(2);
 *tope=f;
 tope++;
 return(f);
}
double Rpn::pop(void)     /* Extrae el elemento superior de la pila */
{
 tope--;
 if (tope<inicio) errorFatal(3);
 return (*tope);
}
```

```
void Rpn::borra(void)           /* Reinicializa la pila */
{
 tope=inicio;
}
int Rpn::toma_instruccion(void)
// Devuelve la instrucción y su atributo
// Los operadores se devuelven como atributo
{
 int i, c;
 /* Mientras sea blanco, \t, o \n no hacer nada */
 while ( (c = toma_char())  == ' ' || c == '\t' || c== '\n' )
      ;
 /* Si no es un dígito devuelve el carácter, un operador */
 if ( c != '.' && (c < '0' || c > '9'))  return(c);
 /* Caso de que sea un número */
 instruccion[0]=c;
 for (i=1; (c=getchar()) >= '0' &&  c <= '9'; i++)
      if (i<MAX_INS)
       instruccion[i]=c;
 if (c=='.')  /* toma la parte fraccionaria */
   {
   if (i<MAX_INS) instruccion[i]=c;
   for (i++; (c=getchar()) >= '0' && c<= '9'; i++)
      if (i<MAX_INS) instruccion[i]=c;
   }
 if (i<MAX_INS) /* El número es correcto */
   {
    devuelve_char(c);
    instruccion[i]='\0';
    return(NUMERO);
   }
 else    /* Es demasiado grande, salta el resto de la línea */
   {
    while (c!='\n' && c!=EOF)
        c = getchar();
    instruccion[MAX_INS-1] = '\0';
    return(GRANDE);
   }
}

int Rpn::toma_char(void)  // Coge si puede un carácter del buffer
{
 return( (pBuffer>0) ? buffer[--pBuffer] : getchar() );
}

void Rpn::devuelve_char(int c)   // Devuelve un carácter al buffer
{
 if (pBuffer>TAM_BUFFER) errorFatal(4);
   else buffer[pBuffer++] = c;
}
```

c. pipo.cpp

```
/*
   pipo.cpp
   Intérprete de la notación polaca inversa
   (reverse polish notation).
   Lenguaje PIPO (PIla_POlaca)
   Versión 1.0, en Pascal, Abril 1988.
   Versión 2.0, en C K&R, Octubre 1990.
   Versión 2.1, en C ANSI, Noviembre 1991.
   Versión 3.0, en C++, Diciembre 1994.
   Versión 3.1, en C++, Diciembre 1998.
```

```
   Este programa es un ejemplo de intérprete puro. El lenguaje PIPO son
   expresiones en notación polaca inversa o RPN (Reverse Polish Notation).
   Un intérprete puro analiza la instrucción y la ejecuta inmediatamente.

   (c) Juan Manuel Cueva Lovelle
       Area de Lenguajes y Sistemas Informáticos
       Departamento de Informática
       Universidad de Oviedo, Diciembre 1998.
*/

#include "rpn.h"

int main (void)
{
 Rpn rpn(100);
 return 0;
}
```

ANEXO XVIII Definición de la máquina abstracta PILAREGI

Se diseña una nueva máquina abstracta PILAREGI con un nuevo repertorio de instrucciones ENSAMPIRE, que permite soportar las ampliaciones definidas en los lenguajes MUSIM/xx. Su denominación se debe a que utilizará pila y registros (máquina de PILA y REGIstros).

La máquina abstracta PILAREGI está compuesta por una zona de memoria para variables y constantes estáticas (*static*), una pila (*stack*), una memoria montón (*heap*), registros y un repertorio de instrucciones. La unidad de manejo de memoria es el *byte*. Los tipos de datos simples que maneja está máquina son: C (char o carácter, ocupa 1 byte), I (integer o entero, ocupa 2 bytes), F (float o real, ocupa 4 bytes), y A (address o direcciones, ocupan 2 bytes, sólo se pueden direccionar 64Kb). Estos números mágicos (la ocupación de cada tipo simple) deben colocarse en todos los programas como parámetros que se puedan modificar con un simple cambio dado que siempre puede ocurrir una evolución de esta arquitectura abstracta.

Además de los tipos simples se definirán otros tipos no simples: H (*handle* o manejador de fichero), S (*struct* o estructura) y U (*union* o unión).

XVIII.1 Memoria estática (*static*)

La memoria para variables estáticas (definidas en tiempo de compilación) son posiciones consecutivas de memoria a partir de la dirección *STATIC*. La unidad de manejo de memoria es el *byte*. Así la primera variable en memoria está en la dirección *STATIC*. La dirección de la segunda variable depende del tipo de la primera:

- C Si el primer dato es un *carácter* (C) el segundo dato está en la dirección *STATIC+1*, pues los datos de tipo carácter ocupan 1 byte.
- I Si el primer dato es un *entero* (I) el segundo dato está en la dirección *STATIC+2*, pues los datos de tipo entero ocupan 2 bytes.
- F Si el primer dato es un *real* (F) el segundo dato está en la dirección *STATIC+4*, pues los datos de tipo real ocupan 4 bytes.
- A Si el primer dato es un puntero, es decir una *dirección* (A), el segundo dato está en la dirección *STATIC+2*, pues los datos de tipo dirección ocupan 2 bytes.
- H Si el primer dato es un manejador de fichero o *handle* (H) el segundo dato está en la dirección *STATIC+2*, pues los datos descriptores de fichero son enteros y ocupan 2 bytes. Los *handles* no son tipos simples de datos, por lo tanto sólo podrán aparecer de tamaño 1, y no se permitirán estructuras de datos que incluyan *handles*. Los valores 0, 1, y 2 están reservados pra teclado, pantalla y errores. El resto de los valores que puede tomar el handle son enteros positivos mayores de 3. Los *handles* de fichero se inicializan a uno de estos enteros, habitualmente se realiza de forma consecutiva a partir de 3.
- S El primer dato también puede ser de tipo *estructura* (S) tal como se comentará más adelante. En ese caso el siguiente dato estará en *STATIC+tamaño de la estructura*, el tamaño de la estructura es la suma de los tamaños de sus miembros.
- U El primer dato también puede ser de tipo *unión* (U) tal como se comentará más adelante. En ese caso el siguiente dato estará en *STATIC+tamaño de la union*. El tamaño de la union es igual al de su miembro mayor.

Aplicando este método consecutivamente se obtienen todas las direcciones de las variables a partir de la dirección *STATIC*. Si hubiera constantes se tratarían de la misma forma que las variables pero inicializadas a un valor. Sólo se pueden inicializar variables de tipo simple (I,C,F) y cadenas de caracteres (arrays de caracteres acabadas con un delimitador que puede ser el carácter '$' o el carácter '\0').

En el caso arrays, cadenas de caracteres el tratamiento es similar pero especificando el tamaño en bytes. Sólo se pueden inicializar los tipos simples y las cadenas de caracteres.

Los ficheros se representan internamente por un entero que es su manejador o *handle*. No se permiten arrays de ficheros, o estructuras de datos de ficheros. Los *handles* de fichero se inicializan a un entero positivo mayor de 3.

Las estructuras se representan por la variable del tipo estructura seguida de una S, y un 1 (estructura simple) o un entero positivo (caso de un array de estructuras con el número de elementos del array). A continuación se coloca la dirección de comienzo de la estructura. En la línea siguiente se colocan los miembros de la estructura como si fueran variables simples, pero con la diferencia de que sus direcciones son desplazamientos desde la dirección de comienzo de la estructura, denominada por el nombre de la variable de tipo estructura. Se permiten estructuras anidadas. También pueden representarse uniones, la diferencia es que los miembros siempre comienzan en la dirección de la unión, y el tamaño de la unión es el de su miembro mayor.

Como ejemplo se presenta el siguiente fragmento de código con la declaración del segmento .STATIC:

```
.STATIC
alba     F  1 STATIC     20.2 ; variable simple de tipo Float no inicializada
hola     C 13 STATIC+4   "HOLA A TODOS$" ; variable de tipo cadena inicializada
alef     I  3 STATIC+17  ? ; array de 3 enteros
jaca     F 20 STATIC+23  ? ; array de 20 reales
pepe     I  1 STATIC+103 ? ; variable simple no inicializada
```

```
punt     A  1 STATIC+105 ? ; puntero
p2       A  5 STATIC+107 ? ; array de 5 punteros
fich     H  1 STATIC+117 3 ; descriptor de fichero
alumno   S  1 STATIC+119 ? ; estructura
nombre   C 20 ALUMNO     ? ; primer miembro de la estructura alumno
dni      C  8 ALUMNO+20  ? ; segundo miembro de la estructura alumno
edad     I  1 ALUMNO+28  ? ; tercer miembro de la estructura alumno
nota     F  1 ALUMNO+30  ? ; cuarto miembro de la estructura alumno
respu    C  1 STATIC+153 'S' ; variable de tipo simple char 153 = 119 + 34
factura  S 10 STATIC+154 ? ; array de 10 estructuras
cliente  C 20 FACTURA ?    ;
nif      C 10 FACTURA+20 ? ;
fecha    S  1 FACTURA+30 ? ; estructura anidada fecha
dia      I  1 FECHA ?      ; primer miembro de fecha
mes      I  1 FECHA+2  ?   ; segundo miembro de fecha
anio     I  1 FECHA+3  ?   ; tercer miembro de fecha
importe  F  1 FACTURA+36 ? ; miembro de factura (36 = 30+6)
siguient A  1 FACTURA+40 ? ; miembro de tipo puntero
q1       A  1 STATIC+196 ? ; variable de tipo puntero (196 = 154+40+2)
pilina   U  1 STATIC+198 ? ; union
entero   I  1 PILINA     ? ;
real     F  1 PILINA     ? ;
letra    C  1 PILINA     ? ;
dime     C  1 STATIC+202 'n' ; variable de tipo simple (202 =198+4)
fich2    H  1 STATIC+203 4 ; descriptor de fichero
fich3    H  1 STATIC+203 5 ; descriptor de fichero

...
```

XVIII.2 Pila o *stack*

La pila comienza en la dirección *STACK0* y el tope de la pila está en la dirección *STACK*. La unidad de manejo de la pila es el byte. Por ejemplo si en la pila se ha introducido un real, la dirección del tope de la pila *STACK* será la dirección *STACK0+4*.

La pila se utilizará para realizar todas las operaciones aritméticas, booleamas, y de comparación. Además se utilizará para realizar la comunicación argumento-parámetro en las llamadas a procedimientos. La pila también contendrá las variables y constantes locales de los procedimientos. La pila también contiene el valor devuelto por un procedimiento.

También se debe introducir en la pila la dirección del código a donde deben de retornar los procedimientos una vez que se han ejecutado.

XVIII.3 Montón o *heap*

La memoria montón o *heap* comienza en la dirección *HEAP0*, contiene los datos y las estructuras de datos definidas en tiempo de ejecución mediante el uso de punteros. La declaración del puntero está en memoria estática, pero el contenido del puntero está en memoria *heap*.

La memoria *heap* crece en dirección opuesta a la memoria *stack*. La dirección *HEAP* apunta al primer bloque de memoria libre. Así si se introduce un real en memoria *heap*, la primera dirección libre *HEAP* será *HEAP0-4*. Si *STACK* o *HEAP* llegan a apuntar a la misma dirección la máquina avisará de agotamiento de memoria heap y stack.

XVIII.4 Código (*code*)

El segmento de código (*code*) contiene las instrucciones de código del programa principal. Cada instrucción es de tamaño fijo y ocupa 16 bytes. El primer byte representa a la instrucción, los siguientes se reparten de distinta forma según los tipos de los operandos. El formato genérico de las instrucciones es el siguiente:

```
<INSTRUCCION>
<INSTRUCCION> <OPERANDO1>
<INSTRUCCION> <OPERANDO1> , <OPERANDO2>
<INSTRUCCION> <OPERANDO1> , <OPERANDO2> , <OPERANDO3>
```

El código del programa principal comienza en la dirección *CODE0* almacenada en el registro *Rcode0*. La instrucción que se está ejecutando está en la dirección *CODE*, contenida en el registro *Rcode*.

El código de los procedimientos comienza en una dirección denominada con el nombre en mayúsculas del procedimiento.

XVIII.5 Registros

Los registros de la máquina abstracta PILAREGI son:

Rstatic Contiene la dirección *STATIC*, de comienzo de la zona de memoria estática a partir de la cual estarán situadas todas las variables estáticas. Este registro *no se puede modificar por el usuario*, tan sólo puede ser utilizado para calcular la dirección de cualquier variable estática. Este registro es de 2 bytes.

Rstack Contiene la dirección *STACK*, que señala siempre al tope de la pila. El contenido de este registro *no puede ser modificado directamente por el usuario*. Hay ciertas instrucciones que manipulan la pila, que lo modifican automáticamente. Este registro es de 2 bytes.

Rstack0 Contiene la dirección *STACK0*, que señala siempre al comienzo de la pila y fin de la memoria *static*. Este registro *no se puede modificar por el usuario*, tan sólo puede ser utilizado por el usuario. Por ejemplo para comprobar que *STATIC* no señala a una dirección inferior a *STACK0*. Cuando la pila está vacia: STACK señala a la misma dirección que STACK0. Este registro es de 2 bytes.

Rheap0 Contiene la dirección *HEAP0*, de comienzo de la zona de memoria heap. No puede ser modificado por el usuario. Este registro es de 2 bytes.

Rheap Contiene la dirección *HEAP*, de comienzo de la zona de memoria heap libre. Debe ser actualizado y manejado por el usuario, la máquina no lo manipula. Puede ser modificado por el usuario. Este registro es de 2 bytes.

Rcode0 Contiene la dirección *CODE0*, de comienzo del código del programa principal. Este registro es de 2 bytes. No se puede modificar por el usuario.

Rcode Contiene la dirección de instrucción en curso *CODE*. Este registro es de 2 bytes. No se puede modificar por el usuario. Inicialmente contiene el valor 0. Se incrementa automáticamente o se modifica por las instrucciones de salto. Ha de tenerse en cuenta que cada instrucción ocupa 16 bytes.

R0 Puede contener cualquier tipo de datos (I, F o C) o una dirección (A). Es un registro de 4 bytes.

R1 Es un registro que puede tener usos múltiples. Puede contener cualquier tipo de datos (I, F o C) o una dirección (A). Es un registro de 4 bytes.

R2 Es un registro que puede tener usos múltiples. Puede contener cualquier tipo de datos (I, F o C) o una dirección (A). Es un registro de 4 bytes.

R3 Es un registro que puede tener usos múltiples. Puede contener cualquier tipo de datos (I, F o C) o una dirección (A). Es un registro de 4 bytes.

XVIII.6 Estructura de las instrucciones de ENSAMPIRE

Las instrucciones en ENSAMPIRE no distinguen mayúsculas de minúsculas, van separadas siempre por un retorno de carro. Los comentarios comienzan por punto y coma (;) y acaban al final de la línea, pueden ocupar una línea o colocarse después de una instrucción, pero nunca en su interior.

El formato de las directivas e instrucciones es el siguiente:

```
<DIRECTIVA>
<INSTRUCCION>
<INSTRUCCION> <OPERANDO1>
<INSTRUCCION> <OPERANDO1> , <OPERANDO2>
<INSTRUCCION> <OPERANDO1> , <OPERANDO2> , <OPERANDO3>
```

Las directivas siempre comienzan por punto e indican comienzo o finalización de un segmento por ejemplo *.EXTERN*, *.STARTIC*, *.CODE*, *.LOCAL*, etc...

Las instrucciones pueden ser sin operandos o tener uno, dos o tres operandos. La coma (,) es el separador de operandos. La lista de las instrucciones posibles se presentan en el apartado *Repertorio de instrucciones*.

Una dirección entre corchetes ([]) representa a su contenido, no a la dirección. Es decir no es la dirección, es a lo que apunta la dirección.

Los caracteres simples están entre apóstrofos(’), las cadenas de caracteres entre comillas ("). Las cadenas de caracteres deben acabar en un delimitador, que será por defecto $, pero también puede ser \0.

XVIII.7 Gramática EBNF de ENSAMPIRE

La estructura de las instrucciones se define mediante la siguiente gramática EBNF:

```
<Programa>                          ::= <Seg_extern>
                                        <Seg_static>
                                        <Seg_código>
                                        { <procedimiento> }
<Seg_extern>                        ::=.EXTERN ’\n’
                                      { <identificador> ’\n’ }
<Seg_static>                        ::=.STATIC ’\n’
                                      { <Declaración> ’\n’ }
<Declaración>                       ::= <identificador> <tipo> <tamaño> <dirección> <valorInicial>
<Tipo>                              ::= I | F | C | A | H | S
<tamaño>                            ::= <const_entera_sin_signo>
<dirección>                         ::= <identificador> <desplazamiento>
```

```
<desplazamiento>          ::= <const_entera_con_signo>
<valorInicial>            ::= ? | <constante>
<Seg_código>              ::= .CODE '\n'
                              { <Instrucción> '\n' }
                              .END
<Procedimiento>           ::= PROC <identificador> '\n'
                               .LOCAL '\n'
                               { <Declaración> '\n'}
                               .CODEPROC '\n'
                               { <Instrucción> '\n'}
                               .ENDPROC
<Instrucción>             ::= <sin_operandos>
                            | <un_operando>
                            | <dos_operandos>
                            | <tres operandos>
<sin_operandos>           ::= <lógicas>
                            | <otras_s_op>
<un_operando>             ::= <aritméticas>
                            | <comparación>
                            | <pila_1op>
                            | <bifurcación>
                            | <llamada_proc>
                            | <cierra_fich>
                            | <libera_heap>
<dos_operandos>           ::= <pila_2op>
                            | <op_MOV_A>
                            | <asigna_heap>
<tres_operandos>          ::= <ope_MOV_I>
                            | <ope_MOV_r>
                            | <ficheros>
<lógicas>                 ::= AND | OR | NOT
<otras_s_op>              ::= ITOF | RETURN
<aritméticas>             ::= <ins_arit> <n_arit>
<ins_arit>                ::= ADD | NEG | MUL | DIV | MOD | EXP
<n_arit>                  ::= 2 | 4
<comparación>             ::= <ins_comp> <n_comp>
<ins_comp>                ::= EQ | GT | LT | GTE | LTE
<n_comp>                  ::= 1 | 2 | 4
<pila_1op>                ::= PUSHA <dirección>
                             | LOAD <n_p2_op>
                             | STORE <n_p2_op>
<bifurcación>             ::= <ins_bifu> <identificador>
<ins_bifu>                ::= GOTO | LBL | JMEQZ | JMLTZ | JMGTZ
<llamada_proc>            ::= CALL <identificador>
<cierra_fich>             ::= CLOSE <enteroSinSigno>
<libera_heap>             ::= FREE <dirección>
<pila_2op>                ::= <ins_p2op_r> <n_p2op> <registro>
                             |  <ins_p2op_c> <n_c> <constante>
<ins_p2op_r>              ::= PUSH | POP
<ins_p2op_c>              ::= PUSHC
<n_p2op>                  ::= 1 | 2 | 4
<n_c>                     ::= <enteroSinSigno>
<op_MOV_A>                ::= MOVRA <registro> , <dirección>
                             |  MOVAR <dirección>, <registro>
<op_MOV_r>                ::= MOV <n_r> , <registro> , <registro>
<n_r>                     ::= 1 | 2 | 4
<asigna_heap>             ::= ALLOC <enteroSinSigno>
<op_MOV_I>                ::= MOVRI <n_MOV_I> <registro>, [ <dirección> ]
```

```
                                    |  MOVIR <n_MOV_I> [ <dirección> ], <registro>
<ficheros>                         ::= OPEN <handle> , <nombre_fich> , <modo>
                                    | READ <handle> , <n_fich> , <registro>
                                    | WRITE <handle>, <n_fich> , <registro>
<handle>                           ::= <const_entera_sin_signo>
<nombre_fich>                      ::= <dirección>
<modo_fich>                        ::= 'r' | 'w' | '+'
<n_fich>                           ::= 1 | 2 | 4
<registro>                         ::= R0 | R1 | R2 | R3
                                     | Rstatic | Rstack | Rstack0 | Rheap
                                     | Rheap0 | Rcode | Rcode0
<constante>                        ::= <const_entera_con_signo>
                                    | <const_entera_sin_signo>
                                    | <'const_char'>
                                    | <const_float_con_signo_sin_exp>
                                    | <"const_cadena">
<const_entera_sin_signo>           ::= <dígito> { <dígito> }
<const_entera_con_signo>           ::= + <const_entera_sin_signo>
                                     | - <const_entera_sin_signo>
                                     | <vacío>
<'const_char'>                     ::= ' <letra> ' | '\<dígito>'
<const_float_con_signo_sin_exp>    ::= <const_entera_con_signo> . <const_entera_sin_signo>
<"const_cadena">                   ::= " {<letra> | <resto_car_menos_"> } $ "
                                    |  " {<letra> | <resto_car_menos_"> } \0 "
<identificador>                    ::= <letra> {<letra>|<dígito>}
<letra>                            ::= _ | a | b | c | d | e | ··· | y | z
<dígito>                           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<resto_car_menos_">                ::= otros
```

XVIII.8 Repertorio de instrucciones

1. Directivas de comienzo y finalización de segmentos

.EXTERN	Indica que los identificadores que se incluyen a continuación son nombres de módulos con procedimientos contenidos en ficheros aparte. El identificador es el nombre del fichero del módulo. Los módulos separados tan sólo contienen el segmento de procedimientos.
.STATIC	Indica que comienza el segmento de definición de las variables y constantes estáticas.
.CODE	Inicio de código.
.END	Indica fin de programa.
.LOCAL	Indica inicio de declaraciones locales a un procedimiento.
.CODEPROC	Indica inicio de código de un procedimiento.
.ENDPROC	Indica fin de procedimiento.

2. Identificadores predefinidos

STATIC	Es la dirección contenida en el registro *Rstatic*. Su valor no se puede modificar.
STACK	Es la dirección contenida en el registro *Rstack*. Su valor se modifica automáticamente por algunas instrucciones. Su valor no se puede modificar directamente, tan sólo a través de las instrucciones de manejo de la pila.
STACK0	Es la dirección contenida en el registro *Rstack0*. Su valor no se puede modificar.
HEAP	Es la dirección contenida en el registro *Rheap*. Su valor debe ser modificado directamente por el programador. Ninguna instrucción la modifica.
HEAP0	Es la dirección contenida en el registro *Rheap0*. Su valor no se puede modificar.
CODE0	Es la dirección contenida en el registro *Rcode0*. Su valor no se puede modificar.
CODE	Es la dirección contenida en el registro *Rcode*. Su valor no se puede modificar.

3. Operaciones inmediatas sobre la pila

PUSHC n const

Coloca la constante de tamaño n bytes en la pila. La pila se incrementa automáticamente en n bytes, es decir el registro *Rstack* se incrementa en n bytes. El valor de n tan solo puede ser: 1, 2, o 4. Sólo hay una excepción en las constantes cadena de caracteres se puede introducir la constante completa, siendo entonces n la longitud de la cadena más uno (el delimitador).
Por ejemplo:
`PUSH 2 25` coloca el entero (2 bytes) en el tope de la pila.
`PUSH 4 30.2` coloca el real 30.2 en el tope de la pila.
`PUSH 1 'p'` coloca el carácter 'p' en el tope de la pila.
`PUSH 15 "PRUEBA CADENAS$"` coloca la cadena en el tope de la pila.

4. Operaciones entre memoria y la pila

PUSHA dirección

Coloca una dirección en la pila. La pila se incrementa automáticamente en 2 bytes, es decir el registro *Rstack* se incrementa en 2 bytes.
Por ejemplo:
`PUSHA HEAP0+50` coloca la dirección `HEAP0+50` en el tope de la pila.
`PUSHA STATIC+25` coloca la dirección `STATIC+25` en el tope de la pila.

LOAD n

Saca de la pila la dirección situada en el tope de la pila e introduce en la pila los n bytes situados en memoria a partir de la dirección dada. El valor de n tan solo puede ser 1, 2 y 4. La pila se incrementa automáticamente en n-2 bytes, es decir el registro *Rstack* se incrementa en n-2 bytes.

STORE n

Saca n+2 bytes de la pila. Los n primeros corresponden a la información que se almacenará en la dirección de memoria dada. Los 2 últimos bytes es la dirección. Despúes de ejecutarse la instrucción a partir de la dirección de memoria d estará la información de n bytes indicada. El valor de n tan solo puede ser 1, 2 y 4. La pila se decrementa automáticamente en n+2 bytes, es decir el registro *Rstack* se decrementa en n+2 bytes.

5. Operaciones entre la pila y los registros

PUSH n , r

Coloca el contenido de los n primeros bytes del registro r en la pila. La pila se incrementa automáticamente en n bytes. El valor de n tan solo puede ser 1, 2 y 4.
Por ejemplo:
`PUSH 2 R0` coloca el entero (2 bytes) contenido en el registro R0 en el tope de la pila.
`PUSH 4 R0` coloca el real (4 bytes) contenido en el registro R0 en el tope de la pila.
`PUSH 1 R0` coloca el carácter (1 byte) contenido en el registro R0 en el tope de la pila.

POP n , r

Saca n bytes del tope de la pila al registro r. La pila se decrementa automáticamente en n bytes. El valor de n tan solo puede ser 1, 2 y 4.
Por ejemplo:
`POP 2 R0` saca un entero o una dirección (2 bytes) de la pila y se almacena en R0.

6. Intercambio de información entre memoria y registros

MOVRI n, r, [d]

Copia el contenido n bytes situados a partir de la dirección de memoria *d* en el registro r. El valor de n tan solo puede ser 1, 2 y 4.

MOVIR n [d], r

Copia n bytes situados en el registro r a partir de la dirección d de la memoria. El valor de n tan solo puede ser 1, 2 y 4.

MOVRA r, d

Copia la dirección d en el registro r.

MOVAR d, r

Copia en el registro r la dirección d.

7. Operaciones entre registros

MOV n, rx, ry

Copia los n bytes primeros del registro ry en rx. Hay que tener en cuenta que el registro destino rx sólo puede ser *R0, R1, R2, R3*, o *Rheap*. El valor de n tan solo puede ser 1, 2 y 4.

8. Operaciones aritméticas

NEG n

Cambia el signo del entero o real contenido en el tope de la pila. El operando tan sólo puede ser 2 o 4. El tamaño de la pila no sufre cambios.

ADD n

Saca de la pila los dos enteros o reales (que supone que hay en el tope y antes del tope) los suma e introduce el resultado en la pila. La pila reduce su tamaño en n bytes. El operando tan sólo puede ser 2 o 4. Por lo tanto la pila habrá disminuido su tamaño en n bytes.

MUL n	Saca de la pila dos enteros lo reales los multiplica e introduce el resultado en la pila. La pila reduce su tamaño en n bytes. El operando tan sólo puede ser 2 o 4. Por lo tanto la pila habrá disminuido su tamaño en n bytes.
DIV n	Opera con los dos últimos enteros o reales introducidos en la pila, extrayendo primero el denominador y después el númerador y dejando en su lugar la división entera o real respectivamente. Por lo tanto la pila habrá disminuido su tamaño en n bytes.
MOD n	Opera con los dos últimos valores (reales o enteros) introducidos en la pila, extrayendo primero el denominador y después el númerador y dejando en su lugar el módulo (entero o real). Por tanto la pila habrá disminuido en n bytes.
EXP n	Saca dos enteros o dos reales de la pila de los cuales el primero es el exponente y el segundo la base, una vez ejecutada la instrucción se introduce en la pila el resultado de elevar la base al exponente. El valor de n es 2 o 4. Por tanto la pila habrá disminuido en n bytes.
ITOF	Saca un entero de la pila e introduce el real equivalente sin decimales. La pila aumenta en 2 bytes.

9. Operaciones de comparación

EQ n	Saca 2n bytes de la pila, y verifica que los n primeros son iguales a los n segundos, en caso afirmativo introduce el entero 1 en la pila, en caso negativo un 0. Los valores de n permitidos son: 1 para C, 2 para I o A, y 4 para F.
	Ejemplo:
	`EQ 2 ; compara la igualdad de dos direcciones o enteros`
	`EQ 4 ; compara dos reales`
	`EQ 1 ; compara dos caracteres.`

GT n	Saca 2n bytes de la pila, y verifica que los n primeros son mayores a los n segundos, en caso afirmativo introduce el entero 1 en la pila, en caso negativo un 0. Los valores de n permitidos son: 1 para C, 2 para I o A, y 4 para F.
LT n	Saca 2n bytes de la pila, y verifica que los n primeros son menores a los n segundos, en caso afirmativo introduce el entero 1 en la pila, en caso negativo un 0. Los valores de n permitidos son: 1 para C, 2 para I o A, y 4 para F.
GTE n	Saca 2n bytes de la pila, y verifica que los n primeros son mayores o iguales a los n segundos, en caso afirmativo introduce el entero 1 en la pila, en caso negativo un 0. Los valores de n permitidos son: 1 para C, 2 para I o A, y 4 para F.
LTE n	Saca 2n bytes de la pila, y verifica que los n primeros son menores o iguales a los n segundos, en caso afirmativo introduce el entero 1 en la pila, en caso negativo un 0. Los valores de n permitidos son: 1 para C, 2 para I o A, y 4 para F.

10. Operadores lógicos

AND	Saca los dos enteros situados en el tope de la pila y realiza la operación lógica AND introduciendo un 0 si es FALSE o un 1 si es TRUE. El cero representa a FALSE y el resto de los valores a TRUE. Por tanto la pila habrá disminuido en 2 bytes. No tiene operando.
OR	Idem que AND pero con OR. No tiene operando.
NOT	Si el entero que hay en el tope de la pila es un cero pasa a tener el valor 1, si tiene un valor distinto de 0 pasa a valer 0. No tiene operando.

11. Instrucciones de salto y bifurcación

GOTO etiqueta	Salto incondicional a la etiqueta. La etiqueta es un identificador.
LBL etiqueta	Define una etiqueta en dicha posición de código.
JMEQZ etiqueta	Saca el entero situado en el tope de la pila y salta a la etiqueta especificada si en el tope había un cero.
JMNQZ etiqueta	Saca el entero situado en el tope de la pila y salta a la etiqueta especificada si en el tope había un valor distinto de cero.

12. Procedimientos

CALL identificador	Llama al procedimiento indicado, los parámetros deben introducirse previamente en la pila. También se introduce en la pila de forma automática la dirección de retorno al código que llamó al subprograma.

PROC identificador	Define el comienzo de un procedimiento. Cada procedimiento tiene un segmento de declaración de variables y constantes locales, y un segmento de código. El segmento de declaración de variables locales es igual que el declarado para el segmento *STATIC* pero en este caso las variables y constantes se definen a partir de la dirección *STACK*. Si hubiera un paso de parámetros por valor también se declararían en el segmento *.LOCAL*.
RETURN	Devuelve el control al programa que llamó al procedimiento. Si se desea devolver algún valor debe utilizarse la pila. Si se ejecuta *RETURN* en el programa principal se finaliza el programa y se devuelve el control al sistema operativo. No tiene operando.

13. Ficheros

OPEN h, dirección, modo	Abre el fichero descrito por el *handle* h, y denominado por la cadena situada a partir de la dirección dada, y en el modo indicado: 'r' para lectura, 'w' escritura y '+' para lectura-escritura.
CLOSE h	Cierra el fichero descrito por el *handle* h.
READ h, n, r	Lee n bytes del fichero descrito por el *handle* h y los introduce en el registro r, donde R es R0, R1, R2, R3. El valor de n tan sólo puede ser 1, 2 o 4.
WRITE h, n, r	Escribe n bytes situados en el registro r en el fichero descrito por el *handle* h. El registro r puede ser R0, R1, R2 y R3.

14. Entrada por teclado y salida a patalla

Es igual que la entrada y salida con fichero pero no es necesario abrir ni cerrar dado que ya existe un *handle* por defecto para teclado y otro para pantalla. También existe otro *handle* por defecto para la salida de errores.

0	Es el *handle* por defecto para lectura por teclado.
1	Es el *handle* por defecto para salida a pantalla.
2	Es el *handle* por defecto para la salida de errores.

15. Gestión de memoria dinámica montón o *heap*

ALLOC n	Reserva n+6 bytes a partir de una dirección d. Téngase en cuenta que las direcciones de memoria heap crecen en sentido inverso. Es decir que si son los primeros n bytes que se reservan el nuevo valor de *HEAP* será *HEAP0-(n+4).* Los 6 bytes extra se reservan en cabeza. Los dos bytes primeros son un entero que almacena el tamaño del bloque reservado n. Los siguientes dos bytes extra indican si el bloque está libre (valor 0) o ocupado (valor 1), con la operación ALLOC se escribe un 1. Los otros dos bytes extras son para guardar otro entero o una dirección auxiliar que pueden ser manejados por el gestor de memoria heap. Por ejemplo para almacenar el contador de referencias o para indicar el bloque al que se tiene que fusionar. La máquina comprueba si *HEAP-(n+4)* es mayor que *STACK*, si fuera cierto devuelve en el tope de la pila la dirección d, en caso contrario devuelve en el tope de la pila un 0.
FREE d	Libera el bloque asignado en la dirección d. Tan sólo coloca el valor 0 en la dirección d+2. La rutina de gestión de memoria *heap* podría realizar liberaciones, fusiones o compactaciones de memoria en el momento que considere oportuno.

ANEXO XIX Lenguaje MUSIM96

XIX.19.1. Especificación léxica

Es indiferente el uso de mayúsculas o minúsculas (al igual que el lenguaje Pascal). Los componentes léxicos del lenguaje se especifican a continuación:

- **Identificadores**: comienzan por una letra y pueden estar seguidos por letras o dígitos. Puede definirse un máximo de caracteres para un identificador no menor de 15.
- **Constantes**: pueden ser enteras, reales sin exponente, de tipo caracter y de tipo cadena. Las constantes de tipo simple se escriben entre comillas simples por ejemplo 'a'. Las de tipo cadena se escriben entre comillas dobles. Por ejemplo "La casa verde". Existe unas constantes de tipo caracter predefinidas '\n', '\0' y '$'. Las dos últimas usadas para delimitar cadenas.
- **Operadores aritméticos**: +, -, *, /, %, ^ y - unario.
- **Operadores de comparación**: <, >, #
- **Operadores lógicos o booleanos**: &, |, !
- **Operadores de manejo de punteros**: & y *
- **Símbolo de la sentencia de asignación**: es el = (igual)

- **Paréntesis, corchetes, punto, llaves, comas, dos puntos y punto y coma.**

  Tipos de datos: INT, FLOAT, CHAR, VOID. No existe el tipo cadena al igual que en lenguaje C, para manejar cadenas se deben utilizar arrays de caracteres o punteros a caracteres. Si existe el tipo constante de tipo cadena.
- **Palabras reservadas**: MAIN, STRUCT, UNION, RETURN, IF, THEN, ELSE, WHILE, FOR, TO, DOWNTO, DO, REPEAT, UNTIL, CASE, OF, END, GOTO, y FILE
- **Procedimientos estándar de entrada/salida**: READ(parámetros), WRITE(parámetros), OPEN(ident_fich, cadena de caracteres o puntero a cadena, modo), CLOSE(ident_fich), READ(ident_fich, parámetros) y WRITE(ident_fich, parámetros).
- **Procedimientos y funciones estándar de gestión de memoria dinámica heap**: función MALLOC y procedimiento FREE (similares a los del lenguaje C).
- **Comentarios**: Comienzan por \ y acaban en fin de línea

XIX.19.2. Especificación sintáctica

Continuando con las especificaciones dadas hasta el MUSIM/6 se añaden:

XIX.19.2.1. Estructuras de control de flujo siguiendo la sintaxis de Pascal estándar

FOR <expr> [TO|DOWNTO] <expr> DO <sent>
IF <expr> THEN <sent> [ELSE <sent>]
WHILE <expr> DO <sent>
REPEAT <sent> UNTIL <expr>
CASE <expr> OF {<valor>: <sent>} END
GOTO cte_entera

XIX.19.2.2. Declaración de funciones prototipo delante de MAIN y construcción de funciones definidas por el usuario detrás de MAIN

Se añaden funciones definidas por el usuario al estilo C pero con la obligatoriedad de tener función prototipo declarada antes de MAIN, y que la construcción de funciones se realiza después de MAIN. Las funciones tienen que soportar recursividad, paso por valor y por dirección, y variables locales. Se incluye el tipo void y la palabra reservada RETURN. En la evaluación de expresiones las funciones tienen la máxima precedencia como las variables y constantes.

XIX.19.2.3. Ficheros secuenciales de texto

Se incorporan las funciones estándar de manejo de ficheros y la palabra reservada FILE para declarar los descriptores de fichero. Las funciones son: OPEN(ident_fich, cadena de caracteres o puntero a cadena, modo), CLOSE(ident_fich), READ(ident_fich, parámetros) y WRITE(ident_fich, parámetros).

XIX.19.2.4. Gestión de memoria dinámica heap y punteros al estilo C

Se introducen punteros a tipos simples y estructuras, con sintaxis igual a C. También se introducen los operadores de manejo de punteros & y *. Las funciones de gestión de memoria son MALLOC y FREE con sintaxis igual al lenguaje C.

XIX.19.3. Especificación semántica

El lenguaje tiene comprobación estricta de tipos al estilo del Pascal estándar.

ANEXO XX El lenguaje Java

El lenguaje Java desarrollado inicialmente por Sun Microsystems en 1993 y actualmente en fase de normalización por comités ANSI e ISO. Este lenguaje anunciado como el lenguaje de Internet, es sin embargo un lenguaje orientado a objetos puro de propósito general, que sirve para desarrollar aplicaciones de cualquier tipo (relacionadas o no con internet). El mayor éxito del lenguaje es que se basa en una máquina abstracta (JVM) independiente del sistema operativo y del ordenador sobre el que trabaja (para una referencia completa véase [ESCU97],[JAVAi]).

XXI.1 Características del lenguaje Java

- *Sencillo*
- *Orientado a objetos puro*
- *Interpretado y compilado*
- *Distribuido y con multihilos*
- *Independiente de la plataforma y portable*
- *Robusto y seguro*

XXI.1.a) Sencillo

- Facil aprendizaje
- No tiene sobrecarga de operadores
- No tiene herencia múltiple
- No tiene punteros explícitos
- Gestión de memoria automática con recolector de basura
- Jerarquía de clases muy rica (gráficos, GUI,...) ya definida

XXI.1.b) Orientado a objetos puro

- Abstracción
- Encapsulación
- Herencia simple
- Polimorfismo y clases abstractas
- Fomenta la reusabilidad
- Todo son clases menos los tipos simples
- No existen funciones libres todo son métodos de clases

XXI.1.c) Interpretado y compilado

- Utiliza como lenguaje intermedio *bytecode*, para la máquina abstracta JVM (Java Virtual Machine)
- El *bytecode* puede interpretarse o compilarse
-

XXI.1.d) Distribuido y con multihilos

- Existe una clase Multihreaded
- Permite múltiples hilos activos concurrentemente
- Clases y primitivas para el manejo de la sincronización

XXI.1.e) Independiente de la plataforma y portable

- Utiliza como lenguaje intermedio *bytecode*, para la máquina abstracta JVM (Java Virtual Machine)
- Compacto (intérprete de 100Kb)
- Verifica el código en tiempo de ejecución

XXI.1.f) Robusto y seguro

- Control estricto de tipos en tiempo de compilación y ejecución
- Restricciones del lenguaje contra la corrupción de memoria
- Recolector de basura
- Integridad de interfaces de usuario

BIBLIOGRAFIA

ABRA94 Abrash M. *Zen of Code Optimization*. Coriolis Group Books, 1994.

AHO86 Aho A.V., R. Sethi and J.D. Ullman. *Compilers: Principles, techniques, and tools*. Addison-Wesley, 1986. Versión castellana: *Compiladores: Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana, 1990.

ALVA91 Alvarez Rojo, A. R. *Traductor de Pascal a C*. Proyecto Fin de Carrera. Escuela Universitaria de Informática de Oviedo, Universidad de Oviedo, 1991.

AMMA81 Ammann, U. "The Zurich implementation". en [BARR81], pp. 63-82.

ANSI74 ANSI (American National Standards Institute). *American National Standard for Information Systems Programming Language COBOL*. X3.23, 1974.

ANSI83 ANSI (American National Standards Institute, U.S. Department of Defense). *American National Standard for Information Systems Programming Language Ada*. ANSI/MIL STD1815A, 1983.

ANSI89 ANSI (American National Standards Institute). *American National Standard for Information Systems Programming Language FORTRAN*. S8 (X3.9-198x) Revision of X3.9-1978. *Fortran Forum*, Vol. 8, N. 4, December 1989.

ANTLi ANTLR en Internet. *http://www.antlr.org*
Herramienta de generación de compiladores.

AZAÑ97 Azañon Esteire, Oscar. *Diseño y construcción de herramientas para la Máquina Virtual Java*. Proyecto Fin de Carrera de la Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón nº9636I, Universidad de Oviedo, 1997.

AZAÑ98 Azañon Esteire, O. y Cueva Lovelle, J.M. "J-- Set of Tools for Native Code Generation for the Java Virtual Machine", *ACM SIGPLAN*, volume 33, number 3, pp.73-79, March 1998.

BAIL90 Bailey R. *Functional Programming with Hope*. Ellis Horwood Ltd., 1990.

BARB86 Barber F., Botti V. J., Pérez T.A (1986). *Introducción a los traductores, compiladores e intérpretes*. Dto. de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia.

BARR81 Barron, D.W. *Pascal- The Language and its Implementation*. John Wiley & Sons, 1981.

BENN90 Bennett, J. P. *Introduction to compiling techniques. A first course using ANSI C, LEX and YACC*. McGraw-Hill, 1990.

BETZ91 Betz D. "Your own tiny object-oriented language". *Dr. Dobb's Journal*, pp. 26-33, September 1991.

BIRD88 Bird R. y P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

BORL91 Borland International Inc. *Borland languages. Open architecture handbook. The Borland Developer's Technical Guide*. Borland (1991).

BORL94 Borland International Inc. *Borland C++ V. 4.02*. 1994

BROD92 Brodersen R.W. *Anatomy of a silicon compiler*. Kluwer Academic Publisers, 1992. Comentado en IEEE Computer, July 1994, pp.117.

BURK46 Burks A.W., Goldstine H.H. y Neumann J. Von. *Preliminary discussion of the logical design of an electronic computing instrument*, Part I, vol. I. Report prepared for U.S. Army Ordenance Department. Institute for Advanced Studies. University of Priceton, 1946.
Documento donde se describe la arquitectura de *Von Neumann*. Reproducido en numerosas publicaciones, en español está reproducido parcialmente y comentado en FERN94.

CABA91 Cabal Montes E., Cueva Lovelle J.M. *Generador de analizadores sintácticos ascendentes: YACCOV*. Cuaderno Didáctico nº 45, Dto. de Matemáticas, Universidad de Oviedo, 1991.

CAST93 Castro J., Cucker F., Messeguer X., Rubio A., Solano L., Vallés B. *Curso de programación*. Ed. McGraw-Hill, 1993.

CHAM92 Chambers C. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Doctoral dissertation, Stanford University (1992).

CHAN90 Chang S. *Visual languages visual programming*. Plenum Press, 1990.

CONW94 Conway D. *Parsing with C++ classes*. ACM SIGPLAN NOTICES, Vol. 29, No. 1, January 1994.

CUEV91 Cueva Lovelle, J. M. *Lenguajes, Gramáticas y Autómatas*. Cuaderno Didáctico nº36, Dto. de Matemáticas, Universidad de Oviedo, 1991.

CUEV93a Cueva Lovelle, J. M. *Análisis léxico en procesadores de lenguaje*. Cuaderno Didáctico nº48, Dto. de Matemáticas, Universidad de Oviedo, 2ª edición, 1993.

CUEV93b Cueva Lovelle, J. M. y Mª P. A. García Fuente. *Programación en FORTRAN*. Cuaderno Didáctico nº 67, Dto. de Matemáticas, Universidad de Oviedo, 1993.

CUEV94a Cueva Lovelle J. M., Mª P. A. García Fuente, B. López Pérez, Mª C. Luengo Díez, y M. Alonso Requejo. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Cuaderno Didáctico nº 69, Dto. de Matemáticas, Universidad de Oviedo, 1994.

CUEV94b Cueva Lovelle J.M., Juan Fuente A. "Diseño y construcción del lenguaje ANIMA OBJECT". *Actas del Congreso Internacional de Ingeniería de Proyectos*, Universidad de Oviedo-Asociación Española de Ingeniería de Proyectos, 1994.

CUEV95a Cueva Lovelle, J.M. *Análisis sintactico en procesadores de lenguaje*. Cuaderno Didáctico nº 61, Dto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1995.

CUEV95b Cueva Lovelle, J.M. *Análisis semántico en procesadores de lenguaje*. Cuaderno Didáctico nº 62, Dto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1995.

CUEV95c Cueva Lovelle, J.M. *Tabla de símbolos en procesadores de lenguaje*. Cuaderno Didáctico nº 54, Dto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1995.

CUEV95d Cueva Lovelle, J.M. *Organización de la memoria en tiempo de ejecución en procesadores de lenguaje*. Cuaderno Didáctico nº 55, Dto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1995.

DAWE91 Dawes J. *The VDM-SL reference guide*. UCL Press (1991)

DEJE89 Dejean D.M. y Zobrish G. W. "A definition optimization technique used in a code translation algorithm". *Comm. of the ACM*, Vol. 32, No. 1, pp. 94-105 (1989).

DEVI93 Devis Botella, R. *Programación orientada a objetos en C++*. Ed. Paraninfo (1993).

DIAZ92 Díaz García, G. "Evaluación del compilador Microsoft C/C++ 7 para MS-DOS y Windows". *Revista Microsoft para programadores*, Vol. 2, Nº 4 (1992)

DIJK68 Dijkstra E. W. "Goto statement considered harmful". *Comm. of ACM, Vol. 11, Nº.3* (1968).

DUNC86 Duncan R. *Advanced MS-DOS*. Microsoft Press, 1986. Versión castellana *MS-DOS avanzado*. Ed. Anaya, 1988.

DUNC93 Duncan R. "Exploring Windows NT executable files". *PC Magazine*, Vol 12, No. 3, pp. 371-380, February 9, 1993. Trae más bibliografís y programas.

ELLI90 Ellis M.A. y Stroustrup B. *The annotated C++ reference manual. ANSI base document*. Addison-Wesley, 1990. Versión castellana: *Manual de referencia C++ con anotaciones. Documento base ANSI.* Addison-Wesley/Díaz de Santos, 1994.

ESCU97 Escudero F. *Java al descubierto.* Prentice-Hall/Sams net (1997).

EVIN94 Evins M. "Objects without classes". *Computer IEEE*. Vol. 27, N. 3, pp. 104-109, 1994.

FERN94 Fernández G. *Conceptos básicos de arquitectura y sistemas operativos. Curso de Ordenadores.* Syserco, Madrid (1994).

FISH94 Fishman S. *Software Development. A legal guide*. Nolo Press, ISBN 0-87337-209-3, 1994.

FRAN91 Franks, N. "Adding an extension language to your software. The little language/application interface". *Dr. Dobb's, pp. 34-43,* September 1991.

GARC91 García Cuesta M.E. y J.M. Cueva Lovelle. *Diseño y construcción de un ensamblador de la familia 80x86*. Cuaderno Didáctico nº43 del Dto. de Matemáticas de la Universidad de Oviedo, 1991.

GARM91 Garmón Salvador Mª M., J.M. Cueva Lovelle, y Salgueiro Vázquez J.C. *Diseño y construcción de un compilador de C (Versión 2.0)*. Cuaderno Didáctico nº46 Dto. de Matemáticas. Universidad de Oviedo, 1991.

GOLD83 Goldberg A. y Robson D. *Smalltalk-80: The language and Its Implementation*. Addison-Wesley, 1983.

GOLD89 Goldberg A. y Robson D. *Smalltalk-80: The language*. Addison-Wesley, 1989.

GOME97a Gómez Vicente, Alberto. *Ccc32 Versión 2.0. Diseño y construcción de un compilador del lenguaje C concurrente para entornos Win32*. Proyecto Fin de Carrera de la Escuela Técnica Superior de Ingenieros Industriales e Ingenieros Informáticos de Gijón nº9622I, Universidad de Oviedo, 1997.

GOME97b Gómez Vicente, Alberto y Cueva Lovelle, Juan Manuel. "Ccc32: Compilador de C concurrente para Win32", *Actas de las VIII Jornadas de Paralelismo*, Universidad de Extremadura, 1997.

GONZ91 González González A. *Construcción de un ensamblador cruzado Z80-80x86*. Proyecto Fin de Carrera, E.U. Informática de Oviedo, Universidad de Oviedo, 1991.

GOOD92 Goodman K.J. "Estudio exhaustivo sobre el código de arranque del Cde Microsoft versión 6.0". *RMP* Vol. 2, No. 2, Abril-Mayo 1992.

GREH94 Grehan R. "Object-Oriented COBOL. The great-granddaddy of procedural languages may soon be stepping out in its new object-oriented shoes". *BYTE*, Vol. 19, No. 9, pp. 197-198 (September, 1994). Versión castellana "La eterna juventud del COBOL. El bisabuelo de los lenguajes basados en procedimientos entra en el mundo de la orientación a objetos". *BYTE España*, Vol. 1, No. 1, pp. 212-214 (Noviembre, 1994).

GRIE71 Gries, D. *Compiler construction for digital computers*. Wiley, 1971. Versión Castellana *Construcción de compiladores*. Ed. Paraninfo, 1975.

HALL93 Hall IV, P.W. "Parsing with C++ constructors". *ACM SIGPLAN Notices*, Vol. 28, No. 4, April 1993.

HAYE94 Hayes F. "Today's compilers", *BYTE*, Vol. 19, No. 2, pp. 76-80, February 1994. Versión castellana: "Los compiladores actuales", *BINARY*, No. 60, pp. 82-86, Abril 1994.

HEKM88 Hekmatpour S., Ince D. *Software prototyping, formal methods and VDM.* Addison-Wesley (1988).

HEND88 Hendrix J.E. *A Small C compiler*. M&T Books (1988).

HOLM95a Holmes J. *Object-oriented compiler construction*. Prentice-Hall, 1995.

HOLM95b Holmes J. *Building your own compiler with C++*. Prentice-Hall, 1995.

HOLU90 Holub A.I. *Compiler design in C*. Prentice-Hall, 1990.

HOLUi Holub en Internet. *http://www.holub.com*

HOPC79 Hopcroft J. E. y Ullman J.D. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

HUSK60 Huskey, H.D., M.H. Halstead, and R. McArthur (1960). "Neliac; a dialect of Algol". *Comm. ACM 20:5*, 350-353.

IEEE91 IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*. IEEE STD 1178-1990, 1991.

JACKi Jack en Internet. *http://www.suntest.com/Jack*

JAVAi Java en Internet. *http://www.javasoft.com*

JENS91 Jensen K. y N. Wirth. *PASCAL. User Manual and Report ISO Pascal Standard*. 4ª Ed. Springer-Verlag, 1991. Versión Castellana de la 2ª Edición: *PASCAL. Manual del usuario e informe*. Ed. El Ateneo, 1984.

JIME92 Jiménez de la Fuente, L. *Traductor de Pascal Orientado a Objetos a Pascal*. Proyecto Fin de Carrera. Escuela Universitaria de Informática de Oviedo, Universidad de Oviedo, 1992.

JOYA94 Joyanes Aguilar, L. *C++ a su alcance. Un enfoque orientado a objetos*. Ed. McGraw-Hill (1994).

JUAN94 Juan Fuente A., *Diseño y construcción del lenguaje ANIMA OBJECT*. Proyecto Fin de Carrera. Escuela Técnica Superior de Ingenieros Informáticos de Gijón, Universidad de Oviedo, 1994.

KATR93 Katrib Mora, M. *Notas de programación orientada a objetos en C++*. Cuaderno Didáctico nº 73, Dto. de Matemáticas. Universidad de Oviedo (1993).

KATR94 Katrib Mora, M. *Programación orientada a objetos en C++*. Infosys, México, 1994.

KELL86 Kelly, M.G. y N. Spies. *FORTH: A Text and Reference*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

KERN71 Kernighan B. W. y L.L. Cherry. "A system for typesetting mathematics". *Comm. ACM, Vol. 18, Nº3*, pp. 151-157, 1971.

KERN78 Kernighan B.W. y D.M. Ritchie. *The C programming language*. Prentice-Hall, 1978. Versión en castellano: *El lenguaje de programación C*, Prentice-Hall, 1985.

KERN84 Kernighan B.W. y Pike B. *The UNIX programming environment*. Prentice-Hall, 1984. Versión castellana: *El entorno de programación UNIX*. Prentice-Hall, 1987.

KERN88 Kernighan B.W. y D.M. Ritchie. *The C programming language*. *Second Edition* Prentice-Hall, 1988. Versión en castellano: *El lenguaje de programación C. Segunda edición*. Prentice-Hall, 1991.

KNUT72 Knuth, D.E. "Ancient babylonian algorithms". *Comm. A.C.M.*, Vol. 15, Nº. 7 (1972).

KRIE83 Krieg-Bruckner B. *et al. Draft reference manual for ANNA: A language for aNNotating Ada programs*. DARPA/RADC, 1983.

LEAV93 Leavenworth B. "PROXY: a scheme-based prototyping language". *Dr. Dobb's Journal*, No. 198, pp. 86-90, March 1993.

LECA78 Lecarme, O. and M.-C. Peyrolle-Thomas. "Self- compiling compilers: an appraisal of their implementation and portability". *Software-Practice and Experience 8*, 149-170, 1978.

LEIV93 Leiva Vázquez J.A. y Cueva Lovelle J.M. *Construcción de un traductor de lenguaje Eiffel a C++*. Cuaderno didáctico nº 76. Departamento de Matemáticas. Universidad de Oviedo (1993).

LEVI79 Levitt K. *et al. The HDM handbook*. SRI International, 1979.

LEVI92 Levine R., Mason T., Brown D. *lex & yacc* . O'Reilly & Associates (1992).

LEWI90 Lewis T. "Code generators". *IEEE Software*, Vol. 7, No. 3, pp. 67-70, May 1990.

LILJ94 Lilja D. J., Bird P. L Eds. *The interaction of compilation technology and computer architecture*. Kluwer Academic Publishers, 1994.

LOWR69 Lowry, E. S. and C. W. Medlock. "Object code optimization". *Comm. ACM 12*, 13-22, 1969.

MART93 Martínez García, J.M. y Cueva Lovelle, J.M. *Generador de analizadores léxicos: GALEX*. Cuaderno Didáctico Nº 66. Dto. de Matemáticas, Universidad de Oviedo, 1993.

MAUR90 Maurer P.M. "Generating test data with enhanced context-free grammars". *IEEE Software*, Vol. 7, No. 4, pp. 50-55, July 1990.

MCAR94 McArthur D.C. "World Wide Web & HTML. Preparing documents for online presentation". *Dr. Dobb's Journal*, No. 224, pp. 18-26. December 1994.

MEYE88 Meyer B. *Object-oriented Software Construction*. Prentice-Hall 1988.

MEYE92 Meyer B. *Eiffel. The language*. Prentice-Hall 1992.

MICR91 Microsoft Corporation. *Microsoft C/C++ Version 7.0*. Microsoft, 1991.

MICR94 Microsoft Corporation. *Microsoft MS-DOS. Programmer's Reference*. Microsoft Press, 1994.

PAEP93 Paepcke, A. *Object-Oriented Programming: The CLOS Perspective*. MIT, 1993.

PAGA91 Pagan F.G. *Partial computation and the construction of language processors*. Prentice-Hall, 1991.

PARE94 Pareja C., Andeyro A., Ojeda M. *Introducción a la Informática. I. Aspectos generales.* Editorial Complutense, 1994.

PATT90 Patterson D.A., Hennessy J.L. *Computer architecture a quantitative approach*. Morgan Kaufmann Publishers (1990).

PIET94 Pietrek M. "Vistazo a PE: repaso del formato de fichero ejecutable de Win32". *RMP*, pp. 41-59, Junio 1994. Trae el programa para visualizar los formatos de Windows NT. Se acompaña de fuente en disquete.

POUN94 Pountain D. "Functional programming comes of age. Beyond Lisp, some new functional programming languages are starting to emerge after the ongoing research of the eighties". *BYTE*, Vol 18, No. 8, pp. 183-184, August 1994. Versión castellana "Más allá de LISP. Los nuevos lenguajes de programación funcional son el resultado de la investigación efectuada a lo largo de los años ochenta". *BYTE España*, Vol. 1, No. 1, pp. 208-212, Noviembre 1994.

PYST80 Pyster A. B. *Compiler design and construction*. Ed. Van Nostrand Reinhold, 1980.

PYST88 Pyster A. B. *Compiler design and construction (with C, Pascal and UNIX tools)*. Second Edition. Ed. Van Nostrand Reinhold, 1988.

REIS92a Reiser, M. *The Oberon System: User Guide and Programmer´s Manual*, Addison-Wesley, 1992.

REIS92b Reiser, M. *Programming in Oberon: Steps beyond Pascal and Modula*, Addison-Wesley, 1992.

RODR87 Rodríguez-Roselló, M.A. *8088-8086/8087 Programación ensamblador en entorno MS DOS*. Ed. Anaya, 1987.

SALG89 Salgueiro Vázquez J.C. y J.M. Cueva Lovelle. *Diseño y construcción de un compilador de C*. Cuaderno Didáctico nº29, Dto. de Matemáticas. Universidad de Oviedo, 1989.

SAMM92 Sammet J.E. "Farewell to Grace Hopper - End of an Era!". *Comm. of ACM*, Vol. 35, No. 4, pp. 128-131, April 1992.

SANC86 Sanchís Llorca, F.J. y C. Galán Pascual *Compiladores: Teoría y construcción*. Ed. Paraninfo, 1986.

SANC89 Sanchez Dueñas, G. y J.A. Valverde Andreu. *Compiladores e intérpretes. Un enfoque pragmático*. Ed. Diaz de Santos, 2ª edición, 1989.

SCHM94 Schmit M. L. *Pentium® Processor Optimization Tools*. AP Professional. 1994.

SCHR85 Schreiner T. A. and Friedman H.G. Jr. *Introduction to compiler construction with UNIX*. Prentice-Hall, 1985.

SELFi Self en Internet. Sobre el lenguaje Self en Internet, puede pedirse información por e-mail en *self-request@self.stanford.edu*. También puede obtenerse documentación por medio de ftp en *self.stanford.edu*, véase el fichero README, y /pub/papers.

SETH89 Sethi R. *Programming Languages. Concepts and Constructs*. Addison-Wesley, 1989. Versión Castellana, *Lenguajes de programación. Conceptos y constructores*, Addison-Wesley Iberoamericana, 1992.

SHAR94 Sharp O., Bacon D. F. "Pruebas de rendimiento. La justa medida", *BYTE España*, Vol 1, No. 1, pp. 76-83, Noviembre 1994.

SIER91a Siering T. "Understanding and using OBJ files". *C Gazette*, Vol. 5, nº 3, Spring 1991.

SIER91b Siering T. "OBJ library management building your own object module manager". *Dr. Dobb's Journal*, pp. 58-62, September 1991.

SITE93 R.L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. "Binary Translation". *Comm. of ACM, Vol. 36, No. 2, pp. 69-81*, 1993.

SLAD87 Slade S. *The T Programming Language*. Prentice Hall (1987).

STEE61 Steel T.B. "A first version of UNCOL". *Western Joint Computer Conference*, pp. 371-378, 1961.

STEE90 Steele Jr. , G.L. *Common Lisp: The Language*. 2nd Ed. Digital Press, 1990.

STER94 Sterling L. y E. Shapiro. *The Art of Prolog. Advanced Programming Techniques*. Second Edition. MIT, 1994.

STEV94 Stevens A. "Borland nonsense: Ready, Aim, Shoot!". *Dr. Dobb's Journal*, Vol. 20, pp. 115-119, April 1994.

STRO86 Stroustrup B. *The C++ programming language*. Addison-Wesley 1986.

STRO91 Stroustrup B. *The C++ programming language*. Second Edition. Addison-Wesley 1991. Versión castellana: *El lenguaje de programación C++*. Addison-Wesley/Díaz de Santos, 1993.

STRO94 Stroustrup B. *The Design and Evolution of C++*. Addison-Wesley 1994.

STRO97 Stroustrup B. *The C++ programming language*. Third Edition. Addison-Wesley 1997. Versión castellana: *El lenguaje de programación C++*. Addison-Wesley/Turpial, 1998.

SWAI94a Swaine M. "Programming Paradigms. Developing for Newton". *Dr. Dobb's Journal*, No. 212, pp. 115-118, March 1994.

SWAI94b Swaine M. "Programming Paradigms. A little RISC lands Apple in the soup". *Dr. Dobb's Journal*, No. 213, pp. 109-113, April 1994.

SWAI94c Swaine M. "Programming Paradigms. Mushroom programming for Newton". *Dr. Dobb's Journal*, No. 215, pp. 105-108, June 1994.

SWAI94d Swaine M. "Programming Paradigms. Mushroom programming the Sequel". *Dr. Dobb's Journal*, No. 216, pp. 107-109, July 1994.

TENN81 Tennent R.D. *Principles of programming languages*. Prentice-Hall, 1981.

TEUF93 Teufel B., Schmidth S., Teufel T. *C^2 compiler concepts*. Springer-Verlag 1993. Versión castellana *Compiladores. Conceptos fundamentales*. Addison-Wesley Iberoamericana, 1995.

THAL90 Thalmann N.M., Thalmann D. *Computer Animation. Theory and Practice*. Springer-Verlag, 1990.

TREM85 Tremblay J. P. and P.G. Sorenson. *The theory and practice of compiler writing*. Ed. McGraw-Hill, 1985.

TSAI94 Tsai L. "Designing an OOP compiler". *The C Users Journal*, Vol 12, No. 5, pp. 37-47, May 94.

UNGA87 Ungar D. and C. Chambers. "Self: The Power of Simplicity", in *Proc. OOPSLA'87 Conf.* ACM Press, pp. 227-241 (1987).

UNGA92 Ungar D., Smith R. B., Chambers C., and Hölzle U. "Object, Message, and Performance: How They Coexist in Self", *Computer IEEE*, Vol. 25, N. 10, pp. 53-64.

VALD91 Valdés R. "Little languages: Big Questions. An informal history of a good idea". *Dr. Dobb's Journal*, September 1991, pp. 16-25.

VARH94 Varhol P. D. "Visual programming's many faces". *BYTE*, Vol. 19, No. 7, pp. 187-188, July 1994.

WAIT85 Waite M. W. and Goos G. *Compiler construction*. Springer- Verlag, second edition 1985.

WATT91 Watt D.A. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

WATT93 Watt D.A. *Programming Language Processors.* Prentice-Hall, 1993.

WAYN96 Wayner P. *Sun gambles on Java chips*. BYTE, November 1996, pp.79-88. Versión española: *Sun apuesta por los chips Java*. BYTE España, Noviembre 1996, pp. 58-65.

WELC91 Welch K.P. "What's in there? Windows 3.0 executable and resource file formats revealed". *Microsoft Systems Journal*, Vol.6 No. 5 Sep/Oct 1991. Versión castellana "Formato de los ficheros ejecutables y de recursos de Windows 3.0". *Revista Microsoft para programadores*, Vol. 1 No. 6, Nov/Dic 1991.

WILH92 Wilhelm R., Maurer D. *Übersetzbau. Theorie, Konstruktion, Generierung*. Springer-Verlag, 1992. Edición francesa *Les compilateurs. Theorie, construction, generation*. Ed. Masson, 1994. Edición inglesa *Compiler Design*, Addison-Wesley, 1995.

WIRT71 Wirth, N."The design of a Pascal compiler". *Software-Practice and Experience 1:4*, pp. 309-333. 1971.

WIRT76 Wirth, N. *Algorithms+data structures= programs*, Prentice-Hall, 1976. Versión Castellana *Algoritmos+estructuras de datos= programas*. Ed. del Castillo, 1980.

WIRT88 Wirth, N. *Programming in Modula-2*, Springer-Verlag, 4ª Ed. 1988.

WIRT92 Wirth, N. y J. Gutknecht. *Project Oberon: The design of an operating system and compiler*. Addison-Wesley, 1992.

Indice

Cuadernos Didácticos
Ingeniería Informática

Nº 1	*Análisis y diseño orientado a objetos*	84-8497-802-8
Nº 2	*Diseño de páginas Web usando HTML*	84-8497-803-6
Nº 3	*Formatos Gráficos*	84-8497-804-4
Nº 4	*Programación en Lenguaje C*	84-8497-805-2
Nº 5	*Administración de Windows NT 4.0*	84-8497-899-0
Nº 6	*Guía del Usuario de Derive para Windows*	84-8497-919-9
Nº 7	*Guía de referencia de MultiBase Cosmos*	84-8416-001-7
Nº 8	*Gestión de un taller con MultiBase Cosmos*	84-8416-034-3
Nº 9	*Ejercicios de Lógica Informática*	84-8416-357-1
Nº 10	*Conceptos Básicos de Procesadores de Lenguaje*	84-8416-889-1
Nº 11	*Introducción a la Administración de Unix*	84-8416-570-1
Nº 12	*Lógica Proposicional para Informática*	84-8416-613-9
Nº 13	*Programación Práctica en Prolog*	84-8416-612-0
Nº 14	*La herramienta CASE MetaCosmos*	84-699-0054-4
Nº 15	*Guía del lenguaje COOL de MultiBase Cosmos*	84-699-0053-6
Nº 16	*Guía de Referencia de Progress*	84-699-2083-9
Nº 17	*Gestión de un Depósito Dental con Progress*	84-699-2082-0
Nº 18	*Guía de Diseño y Construcción de Repositorios con MultiBase Cosmos*	84-699-2081-2
Nº 19	*Gestión y Administración de un Colegio Mayor con MultiBase Cosmos*	84-699-2080-4
Nº 20	*Modelado de software con UML*	84-699-2079-0
Nº 21	*El lenguaje de programación Java*	

Consultor Editorial
Juan Manuel Cueva Lovelle
cueva@lsi.uniovi.es

IMPRIME Y DISTRIBUYE:

Doctor Fleming 3, Local 1
33005 OVIEDO
Tlf. 985250581