

Manual de Desarrollo.

Teoría y Herramientas.

1.- Partes y estructura de un compilador.

Un compilador es una máquina lógica que, a partir de un texto de entrada escrito en un lenguaje de alto nivel (llamado de ahora en adelante texto o programa fuente), genera un código de bajo nivel, más cercano a la máquina, menos intelible por el hombre, pero de más rápida ejecución (al que en lo sucesivo denominaremos código intermedio). Este código intermedio, tras pasar opcionalmente por un proceso de optimización, será convertido en un código directamente inteligible por la máquina (el microprocesador). Este código final generado se llama código objeto.

En algunos compiladores tal vez el código objeto generado sea directamente ejecutable, pero la norma y lo más común es que el código objeto tenga que ser enlazado (*linkado*) con otros módulos de código objeto o con librerías (recopilaciones de código objeto con un formato específico) para resolver referencias desconocidas, como rutinas externas, objetos globales definidos en otros módulos objeto, etcétera. Tras el proceso de *linkado*, el enlazador (que suele ser una herramienta lógica independiente del compilador) genera un código con todas las referencias desconocidas ya resueltas y libre de la información específica que contenía el código objeto. Este código se llama código ejecutable y puede ser (por fin) leído y ejecutado por el microprocesador para el cual se haya desarrollado.

Sobre esas últimas palabras hay que puntualizar algo: un compilador que se ejecuta sobre un procesador X no tiene por qué generar código ejecutable para ese microprocesador, sino que puede dar como resultado un código máquina para un procesador Y. Este tipo de compiladores se denominan compiladores cruzados y su

principal razón de ser es que proporcionan la posibilidad de utilizar un entorno de desarrollo rápido y con buenas herramientas para crear software destinado a ejecutarse en un entorno con pocas facilidades para el desarrollo (seguro que es mejor desarrollar el sistema operativo de un MSX con un compilador cruzado en un sistema UNIX con un procesador muy rápido que en el propio MSX, con un Zilog Z80A muy, pero que muy lento).

Un compilador utiliza un esquema que combina una fase de análisis seguida por otra de síntesis, aunque en la mayoría de las implementaciones las dos fases suelen estar mezcladas.

En la fase de análisis se lee el texto fuente, se divide en fragmentos más fáciles de digerir, se comprueba su validez sintáctica y se realiza un análisis semántico.

La fase de síntesis contiene, al igual que la de análisis, varias subfases: en primer lugar se genera un código intermedio propio del compilador. A continuación, y si el compilador ha sido diseñado con esa capacidad, se optimiza este código intermedio. Por último se genera el código objeto a partir del código intermedio optimizado.

Un compilador se suele dividir en dos partes físicas (aunque estén en el mismo fichero ejecutable): el *front-end* o compilador primario comprende la fase de análisis completa y las dos primeras subfases (generación de código intermedio y optimización) de la fase de síntesis. Por su parte, el *back-end* o compilador secundario comprende la última subfase de síntesis: generación de código objeto.

El compilador primario (*front-end*) es dependiente del

lenguaje fuente, en tanto que el compilador secundario (*back-end*) se construye prestando atención a la máquina para la cual se va a generar código objeto. Esta modularidad permite varias cosas:

- que varios compiladores de lenguajes fuente distintos tengan un compilador secundario idéntico, con la única condición de que el código intermedio que generen sea el mismo.

- que un compilador de un lenguaje tenga varios compiladores secundarios distintos, y pueda generar por consiguiente código objeto para varios microprocesadores diferentes.

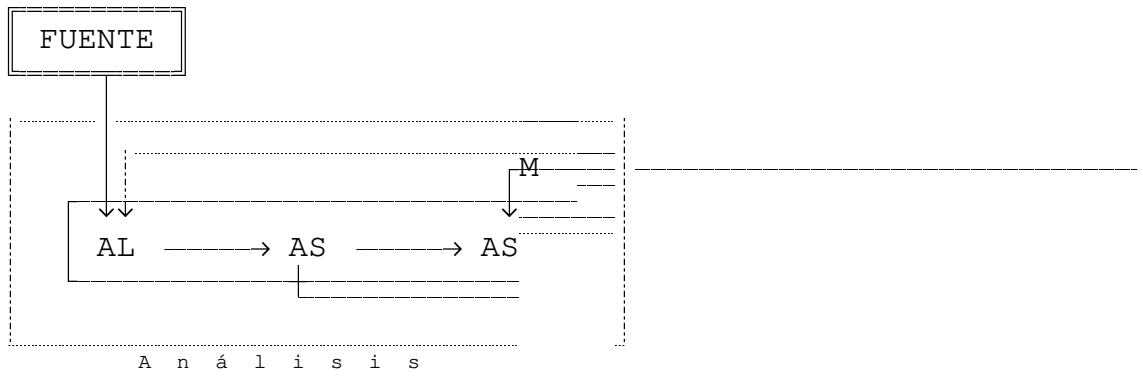
En todos los casos se ahorra un considerable esfuerzo de programación.

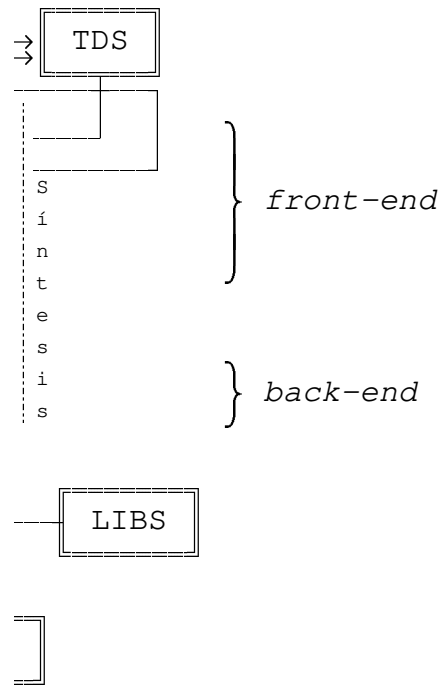
Ya que el compilador secundario es, como se ha dicho, dependiente de la máquina objeto, suele incluir una fase de segunda optimización que tiene en cuenta las características peculiares de ésta (capacidades especiales de direccionamiento, mayor velocidad en determinadas operaciones, etcétera).

Las fases de análisis léxico, análisis sintáctico, análisis semántico y generación de código suelen desarrollarse de forma paralela, interactuando una con otra.

Una estructura fundamental que también forma parte del compilador es la tabla de símbolos. La tabla de símbolos es una serie de estructuras en las cuales se almacena la información (tipo, posición, etc.) de todos los símbolos que se usan en el programa. Un símbolo no tiene por qué ser una variable. Puede ser también una función, una subrutina, una etiqueta, una cadena de caracteres, etc.

En atención a todo lo anteriormente dicho, un compilador se puede representar gráficamente de la siguiente forma:





La simbología utilizada ha sido la siguiente:

- '——→' significa flujo de información en el sentido de la flecha.
- '----->' significa posible flujo de información en el sentido de la flecha.
- Los objetos recuadrados con doble línea son entidades independientes (considero a la tabla de símbolos como una estructura autónoma).
- Las abreviaturas empleadas son:

TDS: tabla de símbolos.

AL: análisis léxico.

AS: análisis sintáctico.

ASM: análisis semántico.
GC: generación de código intermedio.
OC: optimización de código intermedio.
GO: generación de código objeto.
EN: enlazado (linkado).
LIBS: librerías
EJEC: fichero ejecutable.

- Se han señalado las zonas que comprende el análisis, la síntesis, el compilador primario y el compilador secundario.
- Se ha recuadrado el conjunto análisis léxico-análisis sintáctico-análisis semántico-generación de código y se han puesto todos esos procesos al mismo nivel para resaltar que suelen realizarse de forma paralela, interactuando unos con otros de la forma indicada.

A continuación describiremos algo más detalladamente cada una de las fases de la compilación.

1.1.- Análisis léxico o lexicográfico.

El texto de entrada o texto fuente está formado por caracteres (!) generalmente pertenecientes al código ASCII (o, menos frecuentemente, al EBCDIC, más usado en los macroordenadores de IBM). La gramática del lenguaje podría describirse en función de esos caracteres, pero sería una locura de enorme calibre: enter otros terribles problemas, el tamaño de las tablas de reconocimiento de la gramática sería apabullante.

Por esta razón es mucho más lógico describir la gramática en función de una agrupaciones de caracteres que formen palabras clave. Estas agrupaciones se suelen llamar *tokens*, y en adelante

castellanizaré el término (que Dios me perdone) y no lo pondré en cursiva, como el resto de los términos ingleses.

Un token puede ser muchas cosas:

- una palabra reservada del lenguaje: 'volatile' o 'switch' en C, 'division' en COBOL, 'subroutine' en ForTran, etcétera.

- un identificador, que tal vez sea el nombre de una variable o el de una función o, en general, el de un símbolo: 'altura', 'fprintf', 'LeeCharacter' ...

- una secuencia de caracteres o un carácter con significado especial en el lenguaje: '->', '.', '||', '<=', '*', ...

- una cadena de caracteres o un carácter único: 'La altura es %s\n', '?', '\0xfa', etcétera.

- un número entero: 10, 2, 3E12, ...

- un número real: 2.718, 2.12E-13, ...

Y muchas otras cosas.

Un token se identifica por un número que en general suele ser una constante simbólica. Muchas veces, el token ha de llevar asociado un valor: no basta con señalar que el token es una cadena, sino que hay que adjuntar dicha cadena. Idem con los números y varios otros tipos de token.

La parte del compilador que realiza la lectura del texto fuente carácter a carácter y agrupa estos caracteres para formar tokens que luego pasa a la siguiente fase se llama analizador

léxico (en Inglés, *scanner*).

Un analizador léxico es una máquina (generalmente un autómata finito) que reconoce patrones de caracteres. El analizador léxico va recorriendo el fichero fuente de forma lineal. Cada vez que reconoce un token en el fichero, devuelve al programa que lo llamó el código de ese token y la información necesaria sobre el mismo.

Dado que es relativamente complicado construir un analizador léxico para un compilador de mediano tamaño, hay en el mercado una serie de herramientas que permiten la generación de analizadores léxicos a partir de una definición de los tipos de tokens que se quiere reconocer. La más conocida de ellas es Lex. Más adelante hablaremos de Lex.

1.2.- Análisis sintáctico.

La fase de análisis sintáctico sirve para comprobar que la estructura del texto fuente es sintácticamente correcta, es decir, que se ajusta a las reglas de la gramática que define el lenguaje.

Esta fase es llevada a cabo por el módulo del compilador llamado analizador sintáctico o *parser*. Un analizador sintáctico es un algoritmo que, a partir de la formalización de una gramática y de una secuencia de tokens, decide si esa secuencia pertenece o no a la gramática (es decir, al conjunto de las frases que se pueden generar siguiendo las reglas de la gramática).

El analizador sintáctico interactúa casi constantemente con

el analizador léxico: cada vez que necesita un nuevo token, invoca a éste último, el cual le devuelve el código y la información del siguiente token en el fichero de entrada.

La formalización de una gramática es un concepto lingüístico creado por Noam Chomsky (un investigador del Massachusetts Institute of Technology, MIT) a mediados de los años cincuenta como parte de su teoría de lenguajes formales, en la cual unía la Lingüística con la Matemática. Más adelante, J.W. Backus aplicó esta teoría a los lenguajes de programación de ordenadores, durante la creación de la gramática de ForTran (primero) y de ALGOL-60 (después).

Las gramáticas se suelen representar como un conjunto de reglas de derivación que parten de un axioma principal. En este texto utilizaremos la forma normal de Backus (BNF, Backus Normal Form o, a veces, Backus-Naur Form) para describir las reglas de la gramática.

En una gramática se distinguen símbolos terminales y no terminales o metanociones. Los símbolos terminales son los tokens. Los símbolos no terminales son reglas auxiliares de la gramática. El axioma principal es un símbolo no terminal que identifica la unidad de significado a la que se desea llegar. Puede ser una frase en un lenguaje natural, por ejemplo. En un lenguaje de programación, el axioma principal suele ser el fichero fuente (y todos los que éste incluya) o el programa.

Una regla de derivación es una expresión de la forma

$$\text{parte-izquierda} ::= \text{parte-derecha}$$

o

:

parte-izquierda \rightarrow *parte-derecha*

Donde *parte-derecha* es una serie de símbolos terminales y no terminales, y *parte-izquierda* es un único símbolo no terminal (podría ser otras cosas, pero sólo vamos a ver las gramáticas de contexto libre, que requieren esa condición). La lista de símbolos que forma *parte-derecha* puede estar vacía.

Opcionalmente, y como parte de la notación BNF, se pueden indicar varias partes derechas separadas por símbolos '|'.
 '|'

Los símbolos ' ::= ' y ' \rightarrow ' se leen 'se define como' y la regla significa que si tenemos una cualquiera de las secuencias de símbolos que son las partes derechas de la regla de derivación, podemos eliminarla y sustituirla por la parte izquierda. Este proceso de sustitución se va realizando de forma arborescente, hasta que llegamos al axioma o símbolo principal.

Adoptaremos desde ahora la notación de escribir los símbolos no terminales o metanociones en mayúsculas y los terminales en minúsculas y negrita.

Como ejemplo, vamos a mostrar el árbol que se genera al realizar el análisis sintáctico de la secuencia

ddab

perteneciente a la gramática definida por las reglas siguientes:

$$S \rightarrow A \mathbf{b} \quad (1)$$

$$\quad | A \mathbf{c} \quad (2)$$

$$A \rightarrow B \quad (3)$$

$$\quad | C \mathbf{a} \quad (4)$$

$$B \rightarrow \mathbf{y} \quad (5)$$

$$C \rightarrow C \mathbf{d} \quad (6)$$

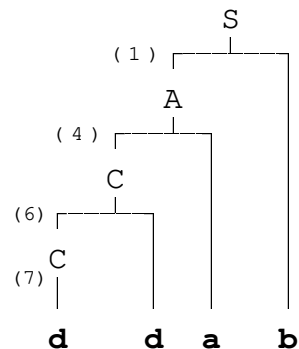
$$| \mathbf{d} \quad (7)$$

Tomamos el primer token, que es una 'd'. Observando las reglas de la gramática, vemos que tenemos la parte derecha de la regla (7). Por consiguiente, sustituimos la 'd' por el no terminal C.

El siguiente token es otra 'd'. Tenemos la cadena C **d**, que es la parte derecha de la regla (6). Sustituimos C **d** por C y pasamos al siguiente token: una 'a'. Como tenemos C **a**, podemos reducir por la regla (4) y obtendremos A.

Al tomar el último token, obtenemos una 'b'. Tenemos la lista de símbolos A **b**, que es la parte derecha de la metanoción S. Reducimos por la regla (1) y nos quedamos con el no terminal S. Dado que ya hemos llegado al axioma principal y que no hay más caracteres en la entrada, aceptamos la cadena y afirmamos que pertenece al lenguaje generado por la gramática.

El árbol de sintaxis que hemos generado en este proceso es el siguiente:



Junto a cada reducción he indicado la regla en virtud de la

cual se ha realizado.

Para este reconocimiento hemos utilizado un algoritmo de reconocimiento de tipo ascendente (un análisis ascendente o *bottom-up*). Vamos ahora a revisar someramente los tipos de análisis sintáctico que se pueden realizar.

1.2.1.- **Análisis descendente.**

Un análisis sintáctico descendente (*top-down*) es aquel que parte del axioma principal de la gramática y va construyendo el árbol sintáctico hacia abajo hasta que llega a los símbolos terminales (las hojas del árbol).

En cada paso se toma un nodo del árbol y, si es un símbolo no terminal, se va sustituyendo por cada una de las reglas que derivan de él (que lo tienen como parte izquierda) hasta encontrar una que se ajuste a la cadena de entrada. Este proceso implica la aplicación del mismo tratamiento a cada uno de los no terminales que haya en la derivación que se esté considerando.

Si ninguna de las derivaciones de la hoja actual es válida, se elimina la hoja actual y se vuelve a la anterior (la de más arriba) para seguir intentándolo.

Hay dos métodos principales para realizar un análisis sintáctico descendente: el descenso recursivo y el análisis LL(1). Veremos someramente cada uno de ellos.

1.2.1.1.- **Descenso recursivo.**

El análisis sintáctico por descenso recursivo es el método más natural e intuitivo. Desgraciadamente, también es el más lento y el más difícil de conjugar con una correcta detección y recuperación de errores sintácticos.

Se parte con el axioma principal de la gramática en la raíz del árbol (que por inescrutables leyes de la informática, y también por simplicidad, se pone arriba del todo) y se invoca al algoritmo Comprobar, pasándole como parámetro el nodo inicial (el axioma).

El algoritmo Comprobar se muestra seguidamente:

```

Algoritmo Comprobar( nodo actual ) devuelve valor lógico
{
  recordar la posición actual de la cadena de entrada.
  repetir lo siguiente para todas las reglas que deriven de
  la metanoción actual, llamando D a la regla derivada:
    {
      coincide ← cierto.
      repetir para todos los símbolos de D, llamando S al
      símbolo tratado:
        {
(C)      si S es un símbolo terminal
(D)      si el token de entrada actual coincide con D
          tomar el siguiente token de entrada.
(E)      si no coincide (esta regla no encaja, tomar la
          siguiente si hay más)
          {
            coincide ← falso.
            salir del bucle de símbolos (ir a H).
          }
(F)      si S no es terminal (es una metanoción)

```

```

(G)          si Comprobar( S ) es falso
              {
                coincide ← falso.
                salir del bucle de símbolos (ir a H).
              }
            } fin del bucle de símbolos.
            si coincide es cierto (la derivación coincide con
            la cadena de entrada)
            retornar cierto.
          } fin del bucle de derivaciones posibles.
(I)  volver la cadena de entrada a su posición inicial.
      retornar falso. (Ninguna de las derivaciones posibles
      encaja con el texto de entrada).
    } fin del algoritmo Comprobar.

```

Este algoritmo (desarrollado de manera formal por mí) es muy parecido a un algoritmo creado por Niklaus Wirth para resolver el problema del salto de caballo. Ambos se basan bastante en algunos conceptos de la teoría de juegos: se crea un árbol y se va intentando llegar al final creando ramas de forma recursiva y borrándolas si no sirven.

Un ejemplo de análisis descendente recursivo es el siguiente:

Analizar la cadena de entrada

ab

para comprobar si pertenece al lenguaje generado por la siguiente gramática:

$$\begin{array}{ll}
 S \rightarrow \mathbf{b} & (1) \\
 \quad | \quad A S & (2)
 \end{array}$$

$$A \rightarrow B \quad (3)$$

$$\quad | \mathbf{a} \quad (4)$$

$$B \rightarrow \mathbf{a} S A \quad (5)$$

Comenzamos invocando al algoritmo Comprobar con el parámetro S.

En $\text{Comprobar}(S)_1$, el bucle (A) deberá considerar las reglas (1) y (2). Tomamos primero la regla (1) y entramos en el bucle (B), que tratará únicamente el símbolo terminal **b**. La condición (C) será cierta, pero la (D) no, ya que el token de entrada es 'a'. Por tanto, ejecutamos (E) y salimos del bucle (B) (el bucle de símbolos). Como (H) no se cumple, hacemos la siguiente iteración del bucle de derivaciones, que proporcionará la regla (2). Los símbolos de esta regla (A y S) serán tratados por el bucle (B). Dado que A es una metanoción, pasamos a (F) y en (G) invocamos recursivamente a $\text{Comprobar}(A)$.

$\text{Comprobar}(A)_1$ entra en el bucle (A), que recorrerá las reglas (3) y (4). Para la regla (3), el bucle (B) tomará sólo el símbolo no terminal B. Este, al cumplir (F), invocará un tercer nivel de Comprobar: $\text{Comprobar}(B)_1$.

$\text{Comprobar}(B)_1$ tiene sólo una regla que comprobar en (A): la regla (5). El bucle (B) tomará los valores **a**, S y A. El primer valor cumple las condiciones (C) y (D), así que tomamos 'b' como nuevo token de entrada y pasamos a la siguiente iteración de (B). El símbolo S causa (a través de (F)) una cuarta llamada a $\text{Comprobar}(S)_2$.

$\text{Comprobar}(S)_2$ toma en el primer valor de (A) la regla (1), lo que motiva que el primer símbolo de (B) sea **b**. Coincide, así que tomamos el siguiente token: el fin de cadena y retornaremos

vía (H) a Comprobar(B)₁.

En Comprobar(B)₁ como el siguiente símbolo tras **a** y S es A, llamamos una vez más a Comprobar(A)₂.

Comprobar(A)₂ tomará la regla (3) en primer lugar. Su primer y único símbolo B motiva, por (F) la llamada Comprobar(B)₂.

Comprobar(B)₂ se da cuenta, al tomar la regla (3) y su primer símbolo **a**, de que este símbolo no coincide con el token actual: fin de cadena. Por lo tanto, salimos del bucle de símbolos y vamos a (H). Como (H) no se cumple, salimos del bucle de derivaciones, ya que la regla B sólo tenía una derivación posible: la regla (5) que acaba de ser descartada. Así las cosas, llegamos a (I), que no hace nada, puesto que la cadena de entrada no se ha modificado (OJO) en esta llamada a Comprobar. (J) devuelve un valor falso a la anterior copia de Comprobar(A)₂.

Esta copia (el Comprobar(A)₂ anterior) tomará la siguiente regla a la (3): la (4) (¡caramba!). El bucle (B) tratará de encajar la **a** y entrará en (C), pero el fallo en (D) terminará con él (esto es como una historia de gánsters). En (I) no modificaremos nada y en (J) volveremos con malas noticias (un valor falso) a Comprobar(B)₁.

El Comprobar(B)₁ ha fracasado estrepitosamente. Ya ha comprobado todas sus reglas (sólo una, la (5)) y éstas a su vez han comprobado sus subreglas. Al final no han conseguido encajar en la cadena de entrada. Por tanto, (I) vuelve a poner la cadena de entrada como estaba antes de entrar en Comprobar(B)₁ (es decir, al principio de la misma) y (J) retorna cabizbajo un valor falso a Comprobar(A)₁.

Comprobar(A)₁ retoma el control y continúa con la regla siguiente a la regla-desastre: la regla (4). Esta regla consta de un único terminal (**a**) que será pasado con éxito por (C) y (D). Así que el nuevo token de entrada será 'b'. Esta regla (4) se ajusta a la cadena de entrada, así que (H) devuelve un valor cierto a Comprobar(S)₁.

Comprobar(S)₁, después de las múltiples andanzas de sus hijos y nietos, recibe la noticia de que su intento de encajar la metanoción A de la regla (2) que estaba intentando ha funcionado. Ahora, en la siguiente iteración de (B), (F) se cumple (ya que el símbolo es S) y (G) llama a Comprobar(S)₃.

Comprobar(S)₃ tomará primero la regla (1), y en (C) y (D) se comprobará que su único símbolo (**b**) encaja con el token de entrada. (H) retornará cierto a Comprobar(S)₁.

De nuevo en Comprobar(S)₁, (G) será cierto, por lo que saldremos del bucle (B), yendo a (H). Como (H) se cumple, retornaremos un valor verdadero que será el resultado definitivo del análisis.

Espero que se comprenda este ejemplo. Para ponerlo algo más fácil, listo a continuación los árboles sintácticos que se han ido creando durante el transcurso del análisis (cfr. Llorca-Galán, p.198):

siempre la metanoción que esté más a la izquierda (*left*) de dicho árbol (segunda 'L'). El uno entre paréntesis puede ser otro número (incluso un cero) e indica que el analizador sintáctico necesita a lo sumo conocer un token de previsión (*look-ahead token*) por delante del actual para poder decidir cuál de las reglas ha de derivar, si hay varias.

En general, estas gramáticas son LL(k), pero las más usadas son las LL(1). Las LL(x) con x mayor que uno son escasas y complicadas. Y hay muy pocos lenguajes que tengan una gramática LL(0) o convertible a LL(0).

El análisis sintáctico descendente LL(1) tiene frente al descenso recursivo la ventaja de que no es necesario el retroceso, que ralentiza demasiado la ejecución del segundo. Gracias al token de previsión o antelación, el analizador sintáctico puede decidir qué parte derecha ha de derivar teniendo la certeza de que, o se termina esa rama recién creada del árbol, o hay un error sintáctico.

El análisis LL(1) utiliza una tabla de reconocimiento que puede asimilarse a una función

$$M \times T \xrightarrow{f} R \cup \{ \lambda \}$$

es decir, es una tabla que tiene como índices de filas todas las metanociones de la gramática, como índices de columnas todos los símbolos terminales y como elementos reglas de derivación o elementos vacíos (λ) que indicarán un error.

Una vez obtenida la tabla de reconocimiento LL(1) mediante un método que incluye el cálculo de varios conjuntos auxiliares, se necesita el algoritmo que utiliza esta tabla para efectuar un reconocimiento (el *driver*) y una estructura tipo pila FIFO que

almacenará símbolos (ya sean terminales o no terminales).

Definamos, antes de ver el algoritmo, tres primitivas:

- PUSH x introduce el símbolo x (que puede ser terminal o no terminal) en la pila.

- POP [x] (x es opcional) saca de la pila el último símbolo introducido en ésta y, si así se especifica, lo almacena en x.

- TOP hace referencia al símbolo de la cabeza de la pila (al último introducido), pero no modifica ésta.

El algoritmo de reconocimiento es el siguiente:

Algoritmo Reconocer.

```

{
  PUSH '$'.
  PUSH axioma.
  Leer token actual.
  repetir lo siguiente mientras token actual sea distinto
    de '$' y TOP sea distinto de '$':
    {
      si TOP es un no terminal
        si tabla [ TOP, token actual ] es  $\lambda$ 
          error sintáctico.
        si no
          (A)      {
                    POP m. (eliminar la metanoción superior de la
                        pila y almacenarla en m).
                    para todos los símbolos de la parte derecha a
                    derivar          (es          decir,          de
                    tabla [ m, token actual ] ) en orden inverso:

```

```
        PUSH simbolo.
    }
si no (TOP es un terminal)
    si TOP es igual al token actual
        { (avanzar/suprimir)
        POP. (eliminar el elemento superior).
        Leer el siguiente token en token actual.
        }
    si no (son distintos)
        error sintáctico.
} fin del repetir.
Aceptar.
} fin del algoritmo Reconocer.
```

Explicaré un poco más claramente la acción que encierran los corchetes (A): si la entrada de la tabla de reconocimiento en la fila indicada por TOP (que es una metanoción) y la columna indicada por el token actual no es un indicador de error (λ), sino una regla de derivación, entonces la aplicamos: sacamos el no terminal de la cabeza de la pila (que coincide con la parte izquierda de la regla) y lo sustituimos por la cadena de símbolos de la parte derecha de la regla. Pero ojo: esta cadena de símbolos ha de escribirse al revés: desde el último símbolo hasta el primero, todos se meten en la pila (PUSH).

Como se ve, la condición de finalización del análisis (me refiero a la aceptación de la cadena de entrada, excepto los errores sintácticos) es que en la cadena de entrada y en la pila haya el mismo símbolo terminal: el indicador de fin de cadena de entrada ('\$'). Por supuesto, esto equivale a decir que tanto la cadena de entrada como la pila estén vacías, ya que en la situación anterior una nueva pasada eliminaría el '\$' de ambos sitios.

Una vez vistos los dos métodos más comunes de análisis descendente, pasaremos a tratar de forma somera el análisis *bottom-up* o ascendente.

1.2.2.- Análisis ascendente.

El análisis sintáctico ascendente o análisis *bottom-up* parte de la secuencia de tokens que constituye la tira de entrada y los va transformando y agrupando en metanociones, subiendo en el árbol sintáctico, hasta llegar al axioma.

Hay varios métodos para realizar un análisis de este tipo. Veremos dos de los más significativos: el método de reducción-desplazamiento y el análisis LR(1).

1.2.2.1.- Análisis por reducción-desplazamiento.

Este análisis necesita para su realización una pila de símbolos. Su forma de actuar es la siguiente:

- Si los símbolos de la cabeza de la pila forman la parte derecha de una regla de producción de la gramática, se sacan de la pila y se sustituyen por la parte izquierda de dicha regla de derivación. A este proceso se le llama reducción por la regla N, donde N es el número de la regla de derivación empleada.

- Si, en cambio, los símbolos iniciales de la pila no forman ninguna parte derecha, se toma el siguiente token de la cadena de entrada y se mete en la pila. A esta acción se la conoce como desplazamiento.

- Este proceso continúa hasta que el axioma quede en la cabeza de la pila.

No vamos a introducirnos más en este método, ya que su uso es poco frecuente. Pasaremos directamente al método LALR(1), más extensamente usado.

1.2.2.2.- **Análisis LALR(1).**

El análisis sintáctico ascendente por el método LALR(1) es un caso particular del más general método LR(1). Una tabla de reconocimiento para un analizador LALR(1) es mucho más pequeña que una para un analizador LR(1).

Al igual que en el análisis LL(1), el número entre paréntesis indica el número de tokens de previsión que utiliza en analizador para resolver conflictos (esta vez los conflictos se deben a la duda entre reducir lo que ya hay en la pila o tomar más símbolos para reducir una metanoción mayor). Las dos primeras letras ('L' y 'A') resaltan la característica del token de avance (*look-ahead token*). La segunda 'L' indica que el analizador recorre la cadena de terminales de entrada de izquierda a derecha (*left to right*) y la 'R' final se debe a que las reducciones se hacen considerando los tokens más a la derecha (*rightmost*) de dicha cadena.

Un analizador LALR (suprimiremos el (1) en adelante) consta de varios elementos:

- Una pila capaz de almacenar estados y símbolos. Se puede dividir en dos pilas, una para cada tipo de elemento, pero por

:

simplicidad y claridad se hace de esta forma.

- Unas tablas de análisis (asimilables a funciones) que indican las transiciones necesarias en cada estado.

- Un programa conductor (*driver*) que lea la secuencia de tokens de entrada y , modificando la pila y consultando las tablas, compruebe su pertenencia al lenguaje.

- Por supuesto, la cadena de entrada.

Lo único que varía de una gramática a otra son las tablas de análisis. El programa conductor permanece inalterado.

La pila almacena una serie de pares símbolo-estado ($S_i e_i$). El elemento superior de la pila es el estado actual. En la configuración inicial, la pila contiene un sólo elemento: el estado inicial de la máquina (estado 0).

Las dos funciones que están representadas por las tablas de análisis son:

- La función F , que acepta un estado de la máquina y un símbolo terminal y genera uno de estos valores (que explicaremos tras ver la función G):

- Desplazar s .

- Reducir x .

- Aceptar.

- Error.

- La función G acepta también un estado de la máquina y un símbolo terminal, y genera un nuevo estado.

El significado de los valores de la función F es el siguiente:

- Desplazar s: se debe tomar el siguiente token de entrada y meterlo en la pila. A continuación se introduce el estado s en la pila.

- Reducir x: x es una regla de la gramática. Se deben sacar de la pila tantos símbolos como haya en la parte derecha de esta regla, incluyendo los estados asociados con cada uno (es decir, se eliminan $2 \cdot n$ elementos de la pila, siendo n el número de símbolos de la parte derecha de la regla x). En la cima de la pila quedará entonces un estado e_x . Ahora metemos en la pila el símbolo que es la parte izquierda de la regla x (llamémosle S_x) y el valor que devuelve la función G llamada con e_x y S_x como parámetros. Es decir, hemos sacado varios pares de símbolos y estados y hemos añadido un símbolo y un estado.

Por ejemplo, si la pila tenía inicialmente la configuración

$$e_0 S_1 e_1 S_2 e_2 S_3 e_3 S_4 e_4 S_5 e_5$$

Para efectuar una reducción por la regla x (consideramos que la regla x es $W \rightarrow S_3 S_4 S_5$) habrá que eliminar de la pila $3 \cdot 2$ elementos, tras lo que tendremos

$$e_0 S_1 e_1 S_2 e_2$$

A continuación, consultamos la tabla (función) G en su

entrada $G(e_2, W)$ y metemos en la pila primero el símbolo W y luego el valor de esa entrada (e_x):

$$e_0 \ S_1 \ e_1 \ S_2 \ e_2 \ W \ e_x$$

- Aceptar: se acepta la cadena de entrada: pertenece al lenguaje.

- Error: se ha detectado un error sintáctico. Se ejecutan los procesos de tratamiento de errores.

El algoritmo de reconocimiento es bien simple: consultar la tabla F con el estado superior de la pila y el símbolo de entrada actual y ejecutar la acción que se indique.

Como ejemplo, reconoceremos la cadena

$$(3 + 1) * 2 \$$$

que pertenece a la gramática de expresiones simples siguiente:

$$E \rightarrow E + T \quad (1)$$

$$| T \quad (2)$$

$$T \rightarrow T * F \quad (3)$$

$$| F \quad (4)$$

$$F \rightarrow (E) \quad (5)$$

$$| n \quad (6)$$

El símbolo terminal n representa un número.

Las tablas de reconocimiento para esta gramática son:

:

Estado	F(e, S)					G(e, S)		
	n	+	*	()	\$	S	T	F
0	d5			d4		1	2	3
1		d6			a			
2		r2	d7		r2 r2			
3		r4	r4		r4 r4			
4	d5			d4		8	2	3
5		r6	r6		r6 r6			
6	d5			d4			9	3
7	d5			d4				10
8		d6			d11			
9		r1	d7		r1 r1			
10		r3	r3		r3 r3			
11		r5	r5		r5 r5			

En esta tabla, rx significa reducir por la regla x; dx significa leer el siguiente token de entrada y meterlo, junto con el estado x, en la pila; a significa aceptar y un espacio en blanco indica un error.

La configuración inicial de la pila es el estado 0.

A continuación se muestran los pasos seguidos para reconocer la cadena de entrada:

:

Pila	Cadena de entrada	Acción
0	(3 + 1) * 2 \$	Desplazar y poner 4.
0 (4	3 + 1) * 2 \$	Desplazar y poner 5.
0 (4 n 5	+ 1) * 2 \$	Red. por 6. G(4,F)= 3
0 (4 F 3	+ 1) * 2 \$	Red. por 4. G(4,T)= 2
0 (4 T 2	+ 1) * 2 \$	Red. por 2. G(4,E)= 8
0 (4 E 8	+ 1) * 2 \$	Desplazar y poner 6.
0 (4 E 8 + 6	1) * 2 \$	Desplazar y poner 5.
0 (4 E 8 + 6 n 5) * 2 \$	Red. por 6. G(6,F)= 3
0 (4 E 8 + 6 F 3) * 2 \$	Red. por 4. G(6,T)= 9
0 (4 E 8 + 6 T 9) * 2 \$	Red. por 1. G(4,E)= 8
0 (4 E 8) * 2 \$	Desplazar y poner 11.
0 (4 E 8) 11	* 2 \$	Red. por 5. G(0,F)= 3
0 F 3	* 2 \$	Red. por 4. G(0,T)= 2
0 T 2	* 2 \$	Desplazar y poner 7.
0 T 2 * 7	2 \$	Desplazar y poner 5.
0 T 2 * 7 n 5	\$	Red. por 6. G(7,F)= 10
0 T 2 * 7 F 10	\$	Red. por 3. G(0,T)= 2
0 T 2	\$	Red. por 2. G(0,E)= 1
0 E 1	\$	Aceptar

El análisis LALR tiene unas buenas características para tratamiento y recuperación de errores. Por ello y por otros motivos se utiliza últimamente con profusión en el desarrollo de compiladores.

Existen en el mercado herramientas de desarrollo que, a partir de la descripción de la gramática, crean un analizador sintáctico para el lenguaje generado por dicha gramática. Una de las herramientas más conocidas de este tipo es Yacc. Estudiaremos este programa posteriormente.

Una vez estudiado el análisis ascendente y el análisis

descendente (y algunos métodos de realización de cada uno), continuamos con la siguiente fase del compilador: el análisis semántico.

1.3.- **Análisis semántico.**

Tras el análisis lexicográfico y el análisis sintáctico, ya sabemos que el texto de entrada está correctamente escrito. Pero ahora tenemos que comprobar que todo lo que se intente hacer sea correcto.

Esta fase interactúa de forma muy estrecha con la tabla de símbolos, añadiendo nuevos símbolos a la misma y consultándola para comprobar la existencia y recabar datos de alguna entidad del programa.

El análisis semántico comprueba, entre otras muchas cosas:

- Que todas las variables utilizadas hayan sido declaradas (si el lenguaje así lo requiere).
- Que no se redefinan variables o módulos.
- Que las declaraciones y la definición de una función o subrutina coinciden tanto en número y tipos de parámetros como en tipo del valor de retorno.
- Que los parámetros que se pasan a las funciones en cada llamada son correctos.
- Que no se intentan operaciones ilegales (producto entre punteros, acceso a matrices con índices no enteros, etcétera).

- Que existan las partes obligatorias (función main en C, módulo PROGRAMA en LCR ...).
- Que ciertas instrucciones estén colocadas en un lugar correcto (los break en C han de estar dentro de un bucle o un switch, y los continue dentro de un bucle).
- Que estén resueltas todas las llamadas a subprocesos y los saltos directos.

1.4.- **Generación de código intermedio.**

La generación de código intermedio es la primera subfase de la fase de síntesis. Su misión es la creación de un programa equivalente al de entrada, pero escrito en un lenguaje especial: el lenguaje intermedio.

El lenguaje intermedio es de más bajo nivel que el lenguaje fuente y tiene características que hacen que los programas escritos en él sean fácilmente optimizables.

También en esta fase se utiliza la tabla de símbolos.

Como lenguajes intermedios más usuales, se pueden citar la Notación Polaca Inversa también llamada RPN (*Reverse Polish Notation*) o notación de Lukasiewicz (el nombre de su creador, el polaco Jan Lukasiewicz), la notación de tercetos, de cuartetos, de tercetos indirectos y el árbol sintáctico.

- La notación polaca inversa se caracteriza por anteponer los argumentos a la operación. Para evaluar una expresión (no

necesariamente aritmética) escrita en RPN se suele usar una pila. Conforme se leen elementos de la expresión, se van apilando si son argumentos. Si se encuentra una operación, se toman los n elementos superiores de la pila (siendo n el número de parámetros requeridos por la operación) y se efectúa la operación sobre esos argumentos. La operación puede, opcionalmente, dejar un resultado sobre la pila.

A veces el número de argumentos para una operación no es fijo (por ejemplo una indexación o una llamada a función). En tales casos, se escribe un argumento que indica cuántos se tienen que tomar de la pila (sin incluirse él mismo). Así pues, una llamada a una función f con los parámetros w , z y t quedará en RPN de la siguiente forma:

`t, z, w, 3, f, llama_a`

La operación `llama_a` sabe que en la cabeza de la pila está el nombre de la subrutina a la que se quiere llamar (f). Toma ese parámetro y lo saca de la pila. A continuación toma el siguiente elemento de la pila (`3`), sabiendo que indica el número de argumentos que se han de pasar a la función. Por último, y tantas veces como indique este último valor, toma un argumento de la función. Obsérvese que los argumentos han de ser metidos en la pila en orden inverso si se quiere su evaluación en el orden correcto.

- La notación en tercetos, también llamada notación de dos direcciones, consta de una serie de elementos compuestos por una operación y hasta dos operandos. Algunas instrucciones requieren el uso de indicadores externos. Por ejemplo, los saltos condicionales se realizan efectuando primero una comparación que actualiza el estado de los indicadores, y saltando luego de forma

condicional al destino. Así, una instrucción del tipo 'salta a x si a es mayor o igual que b' se codificaría en notación de tercetos de la siguiente forma:

```
[ MAY_IG, a, b ]
  [ SALTA_SI_CIERTO, x, - ]
```

El signo '-' indica que no existe ese operando.

- La notación en cuartetos o notación de tres direcciones es muy parecida a la de tercetos, pero con un máximo de tres operandos. La anterior instrucción, en notación de cuartetos, sería:

```
[ SALTA_SI_MAY_IG, a, b, x ]
```

- El código intermedio por tercetos indirectos es una variante en la cual se crea una tabla de tercetos y el código intermedio es una lista de referencias a dicha tabla. Así se ahorra el espacio que de otra forma se gastaría en construcciones duplicadas y se permite una mayor movilidad del código, un requisito muy importante para la optimización.

- La generación de código intermedio en forma de árbol sintáctico es muy adecuada para la optimización de expresiones aritméticas. Consiste en la creación de una serie de estructuras en memoria que representen al programa fuente.

1.5.- Optimización de código.

Una vez generado el código intermedio, éste se somete en algunos compiladores a una fase de optimización.

La optimización puede orientarse a la reducción del tamaño del código o al aumento de la velocidad de ejecución del mismo, aunque generalmente habrá que buscar una solución de compromiso entre ambos objetivos.

Las optimizaciones que se pueden efectuar sobre el código pueden ser varias:

- Uso de funciones intrínsecas: consiste en la sustitución de una llamada a función por el propio código de la función. Se aumenta casi siempre el tamaño del código, pero en según qué casos se puede acelerar bastante la velocidad de ejecución.

- Mejora de las operaciones aritméticas: en casi todos los microprocesadores es menos costoso ejecutar una instrucción de rotación aritmética de bits que una multiplicación o una división. Por otra parte, si toda una expresión se multiplica por cero, ¿para qué calcularla?. Si el microprocesador soporta instrucciones de incremento unitario, ¿por qué usar auto-asignaciones?.

- Desarrollo de bucles: si un bucle tiene pocas iteraciones, en algunos casos puede ser rentable en cuanto a tiempo de ejecución repetir el cuerpo del bucle ese número de veces.

- Compresión de constantes: si una expresión está compuesta únicamente por constantes, o hay varias constantes que pueden operarse entre sí, el compilador puede efectuar esas operaciones y eliminar el trabajo extra que supondría evaluarlas en tiempo de ejecución.

- Eliminación de código innecesario: incluye la eliminación

de objetos declarados pero no referenciados y de código inalcanzable.

- Translación de código invariante inmerso en un bucle: si una instrucción (casi siempre de asignación) se efectúa en el interior de un bucle y esta instrucción no varía durante el transcurso del bucle, se puede sacar fuera del mismo, de tal modo que sólo haya que evaluarla una vez, y no tantas veces como iteraciones tenga el bucle.

Algunas de estas optimizaciones han de hacerse con mucho cuidado, ya que pueden generar efectos colaterales o no considerar algún factor escondido, generando un código ineficiente y la mayoría de las veces incorrecto.

1.6.- Generación de código objeto.

Una vez obtenido y optimizado el código intermedio, éste se ha de convertir en código objeto de una máquina concreta. Esta fase, la última de la síntesis y la única del compilador secundario (*back-end*), puede tener asociada una fase de optimización dependiente de la máquina.

En la optimización orientada a la máquina se trata de mejorar el programa en algunos aspectos que la optimización de código intermedio no ha podido tener en cuenta: instrucciones especiales más potentes, ejecución más rápida de algunas operaciones, etcétera.

Esta fase de generación de código objeto ha de tener muy en cuenta la arquitectura del microprocesador. Hay que considerar, por ejemplo, la ortogonalidad del mismo. Un procesador es tanto más ortogonal cuanto menos especializados son sus registros. Así, si un procesador requiere que todos los accesos indirectos a memoria se realicen mediante un determinado registro, o que todas las multiplicaciones utilicen unos registros fijos, es poco ortogonal.

Los microprocesadores de la casa Intel son bastante poco ortogonales, en contraposición a los de Motorola, que sí son ortogonales.

Una vez detalladas las fases de un compilador, vamos a estudiar someramente las dos herramientas que se han utilizado en la construcción del que se aborda en este proyecto.

La primera utilidad de desarrollo que se comentará será el generador de analizadores léxicos LEX. A continuación se conocerá

al generador de analizadores sintácticos YACC.

2.- **LEX.**

El generador de analizadores lexicográficos LEX es una herramienta que fue descrita por primera vez en un artículo de Michael Lesk y Eric Schmidt. Este último creó la primera versión del programa basándose en ideas de Johnson y Aho. Posteriormente se diseñó FLEX que incorpora un algoritmo de generación más rápido y potente. Tanto LEX como FLEX son dos de las herramientas estándar que se suministran con el sistema operativo UNIX.

Abraxas Software ha creado una versión de LEX (basada en el algoritmo FLEX) para ordenadores personales (PC y MacIntosh) llamada PCLEX. Es de éste programa del que vamos a tratar.

PCLEX es compatible y superior (*upward-compatible*) con LEX y FLEX. Esto quiere decir que las características de estos dos últimos son un subconjunto de las del primero. Los ficheros de descripción de PCLEX son interpretables por LEX y FLEX.

PCLEX es un programa autocontenido: consta de un único fichero ejecutable y no necesita librerías ni otros ficheros fuente.

El cometido de este generador de analizadores léxicos es crear un código fuente en C a partir de un fichero de descripción de patrones. El código C contendrá, además de las tablas de reconocimiento de patrones, una función entera sin parámetros, llamada *yylex*, que leerá el texto de un fichero de entrada e irá devolviendo códigos de tokens conforme vaya encontrándolos. Estos códigos devueltos por *yylex* serán utilizados seguramente por un

analizador sintáctico construido con PCYACC.

El fichero de entrada es una variable externa de tipo FILE * cuyo nombre es **yyin**. Por defecto, está asignado a **stdin**, la entrada estándar. Para reasignarlo, basta con utilizar la función `fopen`.

La estructura de un fichero de definición de patrones es la siguiente:

```
definiciones
%%
patrones
%%
código
```

Tanto el último separador (%%) como la última sección (código) son opcionales.

Veremos estas secciones por separado.

2.1.- Sección de definiciones.

En la parte de definiciones se pueden definir macros que serán luego utilizadas en la sección de patrones. La definición de una macro ha de comenzar en la primera columna y consta del nombre de la macro y la expansión de la macro separada de éste por uno o varios espacios. Una macro se puede expandir en la sección de patrones simplemente utilizando la construcción

```
{ macro }
```

Hay que tener muy en cuenta que PCLEX y FLEX, al expandir

una macro, ponen entre paréntesis esa expansión, en tanto que LEX no lo hace así.

Se puede incluir código C en la parte de definiciones encuadrándolo con los separadores '%{' y '%}':

```
%{  
    código C;  
%}
```

En la sección de definiciones se han de indicar, además, las condiciones de inicio (*start conditions*) exclusivas (*exclusive conditions*) que se vayan a dar en la zona de patrones. Las condiciones de inicio se declaran como

```
%s nombre  
    o  
%Start nombre
```

y las condiciones exclusivas como

```
%x nombre
```

Más adelante veremos qué son las condiciones de inicio y exclusivas.

2.2.- Sección de patrones.

Esta parte del fichero de descripción del analizador está compuesta por una serie de definiciones de patrones, cada uno de los cuales tiene asociada una instrucción C (que, lógicamente, puede ser una instrucción compuesta entre corchetes) que se ejecutará siempre que ese patrón sea encontrado en el texto de

entrada.

Cada patrón ha de estar separado por espacios de su acción C correspondiente. Si en el interior de un patrón se requieren espacios, se usarán secuencias de escape o se entrecomillarán éstos.

Un patrón es una combinación de caracteres y caracteres especiales con el cual pueden coincidir una o más palabras.

PCLEX sólo puede trabajar con el juego de caracteres ASCII puro, es decir, con los códigos 1 a 127 (no admite el 0). Se permiten además las secuencias de escape del C de Kernigham y Ritchie (no del C ANSI): `\n`, `\t`, `\0xx` (número octal), ...

Los caracteres especiales que admite PCLEX son los siguientes:

`" \ [] ^ - ? . * + | () $ / { } % < >`

Explicaremos el significado de cada uno:

- El símbolo comillas dobles (`'"`) sirve para delimitar cadenas literales de texto que han de ser 'encajadas' en el fichero de entrada. Los caracteres especiales pierden sus características (se convierten en caracteres normales) cuando se encierran entre comillas.

- La barra atrás (*backslash*, `'\'`) convierte el carácter que la acompaña en un carácter normal. Por lo tanto, `"["` y `\["` son equivalentes. Por supuesto, también se admite `'\\'`.

- Los corchetes abiertos y cerrados enmarcan una clase de

caracteres: una serie de caracteres de entre los cuales se ha de tomar sólo uno. Así, la expresión [abq] encaja lo mismo con el carácter 'a' que con el 'b' que con el 'q', pero no con combinaciones de éstos.

- El acento circunflejo ('^') sólo tiene significado en dos contextos:

- tras un corchete abierto, es decir, al principio de una clase, niega ésta. Una clase negada encajará con cualquier carácter excepto con los que pertenecen a ella. Por ejemplo, [^aeiou] permitirá cualquier carácter excepto una vocal minúscula.

- al principio de una línea servirá como anclaje de lo siguiente. Sólo se tomará el patrón si se halla a principio de línea. Como ejemplo: ^"#define" encajará la cadena "#define" sólo si es lo primero que aparece en una línea. No aceptará " #define". Ojo: esto no sería correcto en un preprocesador C ANSI. El estándar ANSI indica que una #instrucción del preprocesador puede estar precedida por espacios (cfr. K&R, p. 228).

- El guión ('-') sirve para simplificar las enumeraciones en clases: se expande a los caracteres comprendidos entre el que le precede y el que lo sucede. Así, [a-z] son todas las letras minúsculas y [0-9.] son todos los dígitos y el punto. Para incluirlo en una clase, basta con ponerlo al principio o al final de la misma: tanto [-09] como [09-] comprenden los dígitos 0 y 9 y el guión.

- El símbolo de interrogación cerrada ('?') hace que el elemento que le precede se considere opcional.

- El punto ('.') es un comodín que significa "cualquier carácter excepto un retorno de carro".

- El asterisco indica que el elemento anterior se puede repetir cero o más veces. Por tanto, t[ae]* aceptará cualquier secuencia de 'a'es y 'e's que comience con una 't', incluida la secuencia 't'.

- El signo más ('+') indica posibilidad de repetición de lo anterior una o más veces (una vez como mínimo).

- La barra vertical ('|') hace que se encaje el elemento anterior o el elemento posterior. El patrón pep(ito|ita) aceptará 'pepito' o 'pepita'.

- Los paréntesis agrupan una serie de elementos y hacen que se consideren uno único (para usar, por ejemplo, los operadores +, * o ?).

- El símbolo de dólar ('\$') debe ir al final del patrón y sirve para anclar éste a fin de línea: sólo se aceptará el patrón si tras él hay un retorno de carro.

- La barra de división permite especificar el contexto derecho necesario para la aceptación de un texto. Así, el patrón perro/s aceptará el texto 'perro' sólo si va seguido por una 's'. Ojo: el texto aceptado y recogido será 'perro'. La 's' quedará en el fichero de entrada para la siguiente búsqueda.

- Los corchetes ('{' y '}') cumplen tres misiones:

- Con un número en su interior permiten indicar el número de repeticiones del elemento anterior. El patrón

0{25} es una secuencia de veinticinco ceros.

- Si encierran dos números separados por un coma, definen el número máximo y mínimo de veces que puede aparecer el elemento anterior: [0-9]{1,5} es un número entero de entre una y cinco cifras.

- Si contienen en su interior el nombre de una macro, se sustituyen por el texto de la macro entre paréntesis. Este detalle de los paréntesis difiere de LEX y ha de ser tenido en cuenta.

- El símbolo de porcentaje ('%'), como ya vimos, sirve para la declaración de las condiciones de inicio y exclusivas.

- Los símbolos 'menor que' y 'mayor que' ('<' y '>') determinan las reglas activas con condiciones de inicio o exclusivas. Entre estos dos símbolos se escribirán, separados por comas, el o los nombres de las condiciones de inicio o exclusivas durante el transcurso de las cuales será activo ese patrón o regla.

Tras conocer los caracteres utilizados en la construcción de los patrones, explicaremos qué son las condiciones de inicio (*start conditions*) y las condiciones exclusivas (*exclusive conditions*).

Una condición de inicio es un estado del analizador léxico durante el cual están activas, además de todas las reglas normales, las reglas que pertenecen a esa condición de inicio.

Una regla pertenece a una condición de inicio si tiene como prefijo el nombre de ésta entre ángulos. Una regla puede

pertenecer a más de una condición (ya sea de inicio o exclusiva).

Para entrar en una condición de inicio, se usa la construcción `BEGIN(nombre de la condición)`. Para terminar un estado de este tipo y volver al estado normal, se usa `BEGIN(0)`.

Las condiciones de inicio exclusivas difieren de las normales en que durante el tiempo que el analizador se encuentre en ellas, serán válidas sólo las reglas pertenecientes a esa condición. El resto de las reglas se ignorarán.

Un caso típico de uso de las condiciones exclusivas es el tratamiento de comentarios. El siguiente fragmento de un fichero de descripción sirve para saltar los comentarios de un texto. Consideramos que un comentario comienza por `'{'` y termina por `'}'`, como en Pascal o LCR.

```
"{"          BEGIN( COMENTARIO );
<COMENTARIO>\n  ;
<COMENTARIO>"}"  BEGIN( 0 );
<COMENTARIO>.    ;
```

En la zona de definiciones deberá encontrarse la línea

```
%x COMENTARIO
```

La primera línea detecta el carácter de inicio de comentario y entra en la condición exclusiva `COMENTARIO`. A partir de entonces sólo son válidos los patrones de la segunda, tercera y cuarta líneas.

El patrón de la segunda línea divide los comentarios en

líneas. Esto evita que un comentario demasiado largo llene el buffer del analizador y lo desborde.

En la tercera línea detectamos un carácter de fin de comentario y, usando `BEGIN(0)`, volvemos a las condiciones normales, durante las cuales los patrones `COMENTARIO` no son reconocidos.

En la cuarta línea tomamos cualquier otro carácter que pueda haber dentro del comentario.

Este conjunto de patrones podría haberse sustituido por

```
"{".|\n)*"}"
```

Es decir, por una secuencia de cero o más caracteres cualesquiera (incluidos los saltos de línea) encerrados entre corchetes.

Pero este patrón puede provocar, si se encuentra con un comentario muy largo (o con un comentario no cerrado) un desbordamiento del analizador.

2.3.- Sección de código.

Esta última parte de un fichero de descripción de patrones es opcional, al igual que el separador (%%) que la precede.

Está pensada para que un analizador léxico pueda estar contenido en un único fichero. Todo lo que aparezca después del separador que da entrada a esta sección se copiará literalmente al fichero C generado por PCLEX. Por lo tanto, aquí se pueden incluir las funciones que vayan a ser utilizadas por el analizador léxico e incluso la función main si éste va a constituir un programa autónomo.

Tras estudiar las tres secciones de que consta un fichero de descripción del analizador léxico, vamos a ver cuál es la sintaxis de la línea de comandos de PCLEX.

2.4.- Línea de comandos.

La línea de llamada de PCLEX tiene este formato

```
pclex [opciones] fich-descr
```

El argumento *fich-descr* es el nombre del fichero de descripción de patrones que se quiere traducir a C. Estos ficheros suelen llevar, por convenio, la extensión .l.

Las opciones que acepta PCLEX son las siguientes:

Opción: Significado:

-c llama al fichero de salida yylex.c, en lugar de
 fich-descr.c.

-
- C<fc> llama al fichero de salida *fc*.
 - h muestra información de ayuda sobre las opciones.
 - i genera un analizador que no distingue mayúsculas de minúsculas (*case-insensitive*). Por defecto, las distingue.
 - l no genera directivas *#line*, que pueden confundir a algunos depuradores simbólicos (*CodeView*, *SymDeb*, etcétera).
 - p<fe> utiliza el fichero *fe* como esqueleto (*driver*) del analizador.
 - s si el analizador no puede encajar un texto en ninguno de los patrones, lo imprime en pantalla por defecto. Con esta opción, se detiene con un error.

Con esto se termina el estudio del generador de analizadores lexicográficos PCLEX. Pasamos a estudiar el generador de analizadores sintácticos PCYACC.

3.- YACC.

YACC es un generador de analizadores sintácticos ascendentes de tipo LALR(1). Traduce un fichero de descripción de gramática (*grammar description file*, GDF) en un programa C que contiene, además de las tablas de análisis, una función entera sin parámetros (llamada *yyparse*) que analiza la sintaxis definida por esa gramática.

YACC fue desarrollado en 1975 en los laboratorios Bell de AT&T (lugar donde también nacieron C y UNIX) por Stephen C. Johnson y actualmente es, junto con LEX, una herramienta estándar del sistema operativo UNIX. PCYACC es una implementación de YACC

para ordenadores personales, creada por Abraxas Software.

El formato del fichero de descripción de la gramática que constituye la entrada de PCYACC es muy similar al de PCLEX:

```
declaraciones
%%
reglas
%%
código
```

Al igual que en PCLEX, tanto el separador que inicia la sección de código como la propia sección de código son opcionales.

Trataremos una por una las secciones.

3.1.- Sección de declaraciones.

La primera sección de un fichero de descripción de gramática recoge las declaraciones que serán necesarias para la correcta generación del analizador.

En esta zona se puede insertar código C delimitándolo, al igual que en PCLEX, con los separadores %{ y %}.

La principal declaración que se ha de hacer es la de la unión que contendrá los atributos asociados a cada símbolo de la gramática. Esta declaración se puede hacer de dos formas:

- declarando en C un tipo llamado YYSTYPE que sea la unión que se usará como elemento de la pila de valores (o atributos)

- usando la instrucción `%union`, seguida de la definición de la unión en sintaxis C.

Han de declararse también todos los símbolos terminales (tokens) que se utilizarán en la gramática, excepto aquellos que consten de un sólo carácter, que pueden incluirse literalmente en las reglas sin previa declaración.

Esta declaración se hace usando la instrucción `%token`, seguida por los tokens que se vayan a declarar separados por espacios. Se pueden usar varias intrucciones de este tipo.

Algunos símbolos de la gramática (terminales o no) tendrán asociado un valor o atributo, que se usará para pasar información entre las reglas. La asignación de un tipo de atributo a un símbolo de la gramática se realiza mediante la instrucción `%type`, cuya sintaxis es la siguiente:

```
%type <campo> símbolo [símbolo ...]
```

El argumento *campo* es el nombre del campo de la unión que se usará como tipo de atributo de los símbolos listados.

Se pueden también usar varias instrucciones de este tipo.

Las instrucciones de definición de precedencia y asociatividad son `%left`, `%right` y `%noassoc`. Seguidas por una lista de símbolos o identificadores separados por espacios, otorgan a esos símbolos la asociatividad que indica cada instrucción (izquierda, derecha y ninguna, respectivamente).

La precedencia de un símbolo es mayor cuanto más tarde aparezca su definición de precedencia. Así, en las declaraciones

```
%left '+' '-'  
%left '*' '/'  
%left MENOS_UNARIO
```

los símbolos '+' y '-' tienen igual precedencia y se evalúan de izquierda a derecha. Los símbolos '*' y '/' tienen más precedencia que los anteriores y también se evalúan de izquierda a derecha. El símbolo con mayor precedencia es MENOS_UNARIO, que tiene el mismo orden de evaluación. Este último no es propiamente un símbolo, sino un indicador de precedencia.

Cada regla de la gramática tiene asociada una precedencia que es la del último símbolo terminal de su parte derecha. Si dicho símbolo no tiene asignada una precedencia, la regla no tiene precedencia.

Se puede asignar a una regla una precedencia mediante la instrucción %prec.

Las relaciones de precedencia y asociatividad sirven para resolver conflictos. Veremos más adelante cómo se lleva a cabo este proceso.

La última declaración que se puede incluir en esta zona del fichero de descripción de gramática es la instrucción %start. Seguida por un símbolo no terminal, lo designa como axioma de la gramática. Si no existe esta declaración se considera que el axioma es la primera regla que se encuentre en la zona destinada a ellas.

3.2.- Sección de reglas de la gramática.

En esta sección se enumeran las reglas de derivación que componen la gramática. Las reglas se especifican en una variante de la notación BNF (Backus-Naur Form): la parte derecha se separa de la izquierda mediante un símbolo de dos puntos (':'). Las distintas alternativas se separan mediante barras verticales ('|') y cada producción se ha de terminar con un punto y coma(';').

```
parte izquierda : Pd1
                 | Pd2
                 | Pd3
                 ...
                 | Pdn
                 ;
```

Los elementos *Pdx* son las distintas partes derechas alternativas. Una parte derecha puede estar vacía.

En las partes derechas de las reglas de la gramática se pueden intercalar fragmentos de código entre corchetes, que forman las llamadas acciones semánticas. Las acciones semánticas se consideran como símbolos y se ejecutan cuando son reducidas.

La parte derecha de una regla está, pues, formada por una serie de símbolos terminales, símbolos no terminales y acciones semánticas. Los tokens (símbolos terminales) de un sólo carácter de longitud se pueden poner entre apóstrofes y no tienen por qué declararse en la sección anterior.

Dentro de una acción semántica nos podemos referir al atributo de cualquier símbolo que la preceda en la parte derecha (si es que el símbolo tiene asignado un atributo) usando la

notación \$x\$, donde x es el número de orden del símbolo en la parte derecha (como dije, las acciones semánticas también se cuentan).

El término \$\$ se refiere al atributo del símbolo que forma la parte izquierda de la regla. Un ejemplo:

```
operación: opdo '+' { printf("más "); } opdo { $$= $1+$4; }
;
```

```
opdo: NUMERO { $$= $1; printf (" %d ", $1 ); }
;
```

Para que estas reglas sean correctas, en la sección de declaraciones ha de haberse definido el token NUMERO, y los símbolos NUMERO, opdo y operación han de tener un tipo entero asignado como atributo. El atributo del símbolo terminal NUMERO lo rellenará el analizador léxico como veremos pronto.

Una gramática LALR puede contener conflictos, que serán detectados y notificados por PCYACC. Los conflictos que pueden aparecer en una gramática son conflictos reducción/reducción y conflictos desplazamiento/reducción.

3.2.1.- Conflictos reducción/reducción.

Un conflicto reducción/reducción se da cuando hay varias posibilidades de reducción y el token de antelación o previsión (*look-ahead token*) no proporciona al analizador sintáctico suficiente información para decidir cuál de varias reglas se ha de reducir.

Hay pocos ejemplos de conflictos reducción/reducción.

Cuando se da un conflicto de este tipo es que la gramática está mal construida. PCYACC, por defecto, soluciona estos conflictos aplicando para reducción la regla que primero aparezca en la gramática. Pero lo mejor es revisar la gramática y tratar de modificarla para resolver el conflicto.

3.2.2.- Conflictos desplazamiento-reducción.

Estos conflictos, mucho más frecuentes que los de reducción/reducción, aparecen cuando el analizador tiene un estado en el que no sabe si reducir lo que ya hay en la pila (puesto que se ajusta a una parte derecha) o seguir desplazando (leyendo tokens de entrada), puesto que hay una regla que contiene lo que ya hay en la pila y algunos símbolos más.

El ejemplo más claro y típico de este conflicto se presenta en la mayoría de las gramáticas de los lenguajes de programación: es el problema de los ELSE.

En una gramática, las dos reglas siguientes

```
instr: IF expr THEN instr
      | IF expr THEN instr ELSE instr
      ;
```

generan un conflicto desplazamiento/reducción. Cuando el analizador reduce el primer no terminal instr, tiene en la pila la cadena de símbolos 'IF expr THEN instr'. Ahora se presenta una alternativa: ¿se debe reducir esa cadena y convertirla en la metanoción instr o por el contrario hay que seguir desplazando para tratar de obtener los símbolos 'ELSE' e 'instr'?

PCYACC resuelve estos conflictos decidiéndose siempre por el desplazamiento. Así se cumple la regla de la mayoría de los lenguajes de que cada IF está emparejado con su ELSE más próximo.

Pero a veces no se desea que se tome esa acción por defecto, y se quiere que el analizador opte por la reducción. Para ayudar al control de estas acciones se usan las reglas de precedencia y asociación de la siguiente forma:

- En un conflicto reducción/reducción o desplazamiento/reducción, si ninguna de las reglas implicadas tiene asociada una precedencia se efectúan las acciones por defecto.

- Si en un conflicto desplazamiento/reducción las reglas tienen asociadas precedencia y asociatividad, se hace lo siguiente:

- Si la precedencia dl token de entrada es mayor que la de la regla, se efectúa un desplazamiento.

- Si es menor, se lleva a cabo la reducción.

- Si son iguales, manda la asociatividad de la regla:

- Si es a izquierdas, se reduce.

- Si es a derechas, se desplaza.

- Si no es asociativa, se genera un error.

3.3.- Sección de código.

Todo lo que se encuentra en esta sección es directamente escrito en el fichero de código C generado por PCYACC.

Con esta sección acabamos el estudio de la estructura del fichero de descripción de la gramática. Estudiaremos seguidamente las funciones que necesita obligatoriamente el analizador generado por PCYACC.

3.4.- Funciones auxiliares obligatorias.

Estas funciones son básicamente dos: la función `yylex` y la función `yyerror`.

- La función `yylex` es una función entera sin parámetros

```
int yylex( void );
```

que, en cada llamada, devuelve el código del token que lea del fichero de entrada. Esta función suele ser generada por PCLEX, aunque nada impide construir una a mano.

La función `yylex` se ocupa también de almacenar en el campo correspondiente de la unión global `yyval` el atributo asociado al token que devuelve si es necesario. Así, al reconocer el analizador léxico un número entero, antes de retornar la constante `ENTERO` (por ejemplo), almacenará el valor del número en el campo correspondiente de la unión `yyval`.

La función `yylex` es llamada por el analizador sintáctico cada vez que éste necesita un nuevo token.

- La función `yyerror` es una función sin valor de retorno que acepta un puntero a carácter (una cadena de caracteres):

```
void yyerror( char * );
```

Esta función es llamada cada vez que se produce un error. La cadena contiene el texto del error.

3.5.- Línea de comandos.

La línea de comandos de PCYACC tiene el siguiente formato:

```
pcyacc [opciones] fdg
```

Donde *fdg* es el nombre del fichero de descripción de gramática que queremos que PCYACC procese. Estos ficheros suelen tener la extensión *.y*.

Las opciones que acepta PCYACC son las siguientes:

Opción: Significado:

- c llama al fichero de salida *yytab.c*, en lugar de *fdg.c*.
- C<*fc*> llama al fichero de salida *fc*.
- d genera un fichero llamado *yytab.h* que contiene los `#defines` de los tokens de la gramática y la declaración de la unión, entre otras cosas.
- D<*fh*> genera el mismo fichero, pero con el nombre *fh*.
- h muestra una pantalla de ayuda sobre las opciones.
- p<*fe*> utiliza el fichero *fe* como esqueleto (*driver*) del analizador.
- s hace que los elementos de las tablas generadas

:

- sean de tipo short int, en lugar de int.
- S revisa únicamente la sintaxis del *fdg*, sin generar nada.
 - t construye el analizador de tal modo que genere un árbol de reconocimiento y lo almacene en el fichero *yy.ast*
 - T<*fa*> igual que -t, pero almacena el árbol en *fa*.
 - v genera una tabla de información sobre los estados del analizador en el fichero *yy.lrt*.
 - V<*ft*> genera la misma tabla en el fichero *ft*.
 - n no genera directivas *#line*, que pueden confundir a algunos depuradores simbólicos (*CodeView*, *SymDeb*, etcétera).
 - r va informando de los pasos que realiza en la generación.

Con esto se da por terminada la explicación de PCYACC.

Una vez dada una introducción teórica y vistas las herramientas que se van a usar en el desarrollo de este proyecto, pasaremos a especificar el diseño del compilador.