Capítulo 1. Lenguajes de programación

Alfabetos, palabras, lenguajes

Alfabeto: un conjunto finito no vacío A.

Letra: un elemento de un alfabeto.

Palabra: una sucesión de letras.

Longitud de una palabra: |x| = número de letras.

Palabra vacía: palabra de longitud 0.

Universo del discurso o lenguaje universal: conjunto de todas las palabras sobre un alfabeto. Conjunto infinito, numerable, W(A).

Concatenación de palabras. Propiedades (cerrada sobre W(A), asociativa, elemento neutro, monoide libre no abeliano, |xy|=|x|+|y|.

Cabeza y cola de una palabra.

Potencia de una palabra: |x*i|=i.|x|.

Reflexión de palabras: |x*-1|=|x|.

Lenguaje: L c W(A).

Unión de lenguajes: cerrada, asociativa, elemento neutro, conmutativa, idempotente.

Concatenación de lenguajes: cerrada, asociativa, elemento neutro.

Binoide libre.

Potencia de un lenguaje. Clausura positiva. Iteración o cierre.

Reflexión, intersección, complementación.

Gramáticas

G = (At, An, S, P)

Producción: (x,y) representado x::=y.

Derivación directa mediante x::=y : v=>w <=> existen z,u v=zxu, w=zyu.

Derivación: v =>+ w.

Relación de Thue: v =>* w.

Notación de Backus.

Forma sentencial de G: S =>* x.

Sentencia (instrucción) de G: S =>* x y x está en At*.

Lenguaje asociado a G: { x | S =>* x }.

Frase de una forma sentencial: S =>* xUy, U =>+ u.

Frase simple de una forma sentencial: S =>* xUy, U => u.

Asidero: frase simple más a la izquierda.

Recursividad de una gramática: uUv =>+ xUy.

Recursividad de una regla: uUv ::= xUy.

Recursividad a izquierdas y a derechas.

Clasificación de Chomsky y de los lenguajes. Máquinas. Equivalencias.

Arbol de derivación. Subárbol.

Ambigüedad de sentencias, gramáticas, lenguajes

Lenguajes naturales

Todos representables mediante gramáticas independientes del contexto, excepto dos:

- Alemán suizo:

  Juan vio a Luis dejar que María ayudara a Pedro a hacer que Felipe trabajara.

  Juan Luis María Pedro Felipe vio dejar ayudar hacer trabajar.

  Algunos verbos exigen acusativo, otros dativo. Supongamos que los acusativos fueran primero, después los dativos: tenemos palabras de la forma

  A*n.B*m.C*n.D*m

  donde A = frase nominal en acusativo, B = frase nominal en dativo, C = verbo que exige acusativo, D = verbo que exige dativo.

  Este lenguaje no es independiente del contexto.

- Bambara.

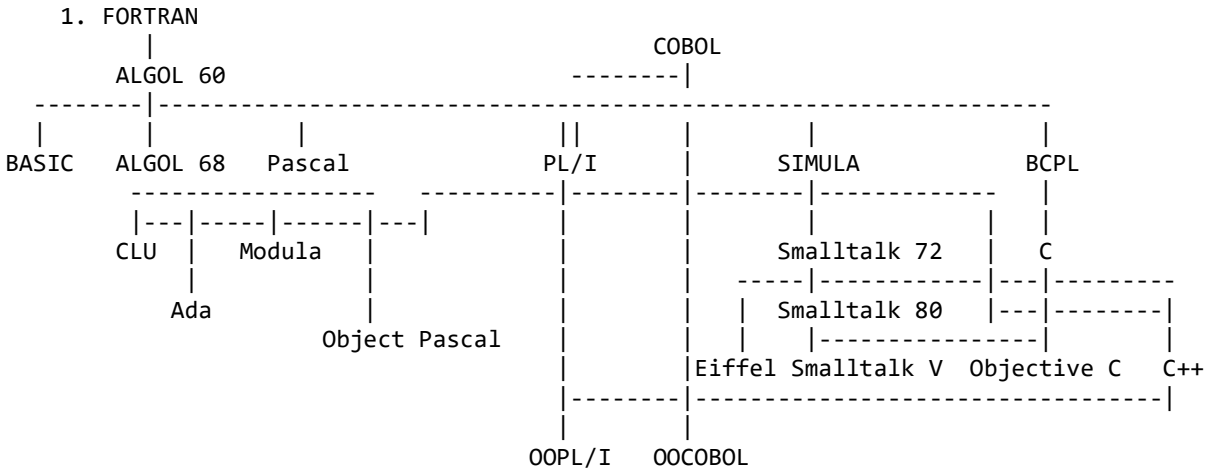
Lenguajes de programación

Generaciones de lenguajes

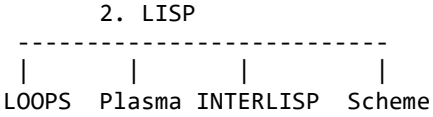
- 1830: Condesa Ada Lovelace
- Años 40: Prehistoria: programación con cables
- 1945-55: Primera generación: lenguajes de la máquina
- 1950- : Segunda generación: lenguajes simbólicos
- 1958- : Tercera generación: lenguajes de alto nivel
- ? : Cuarta generación: "frameworks" ?

Lenguajes de alto nivel

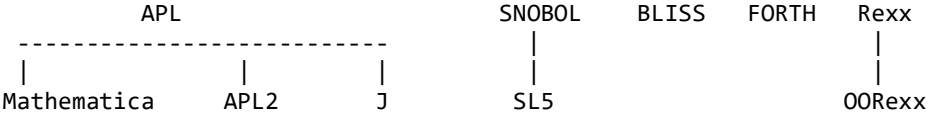
Lenguajes imperativos



Lenguajes aplicativos o funcionales



Otros lenguajes



Uso de los lenguajes

	1980
COBOL	47.3%
RPG	14.8%
FORTRAN	8.9%
Simbólicos	8.4%
BASIC	5.4%
PL/I	4.8%
Pascal	1.2%
Otros	8.9%

Traductores

- Ensamblador.
- Compilador.
- Intérprete.
- Compilador-intérprete.
- Translator Writing System (Compilador de compiladores).

Símbolos.

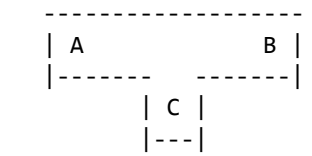
Técnica del Bootstraping.

Partes de un traductor

- Tablas de símbolos.
- Análisis morfológico.
- Análisis sintáctico.
- Análisis semántico.
- Generación de código.
- Gestión de memoria.
- Recuperación de errores.
- Optimización de código.

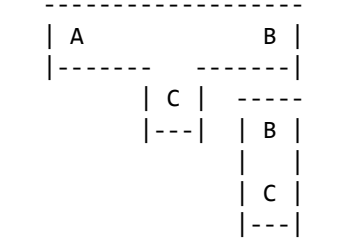
Traductores (símbolos)

- Ensamblador.
- Compilador.

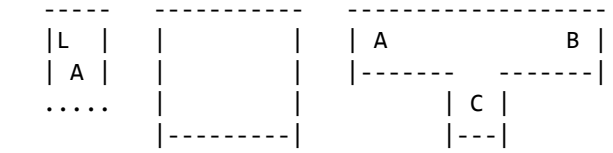


- Intérprete.

- Compilador-intérprete.

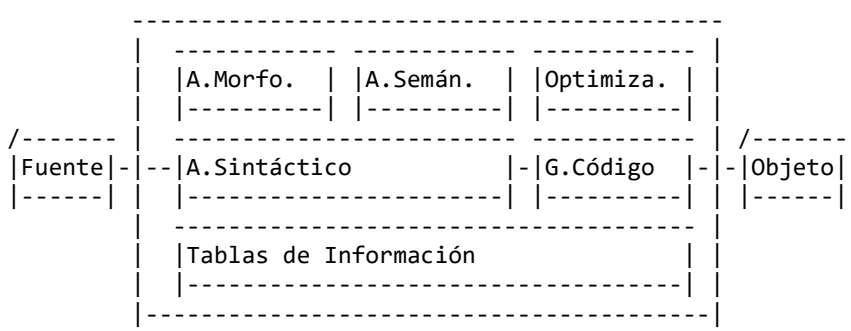


- Translator Writing System (Compilador de compiladores).



Partes de un compilador

- Tablas de símbolos
- Análisis morfológico
- Análisis sintáctico
- Análisis semántico
- Generación de código
- Gestión de memoria
- Recuperación de errores
- Optimización de código

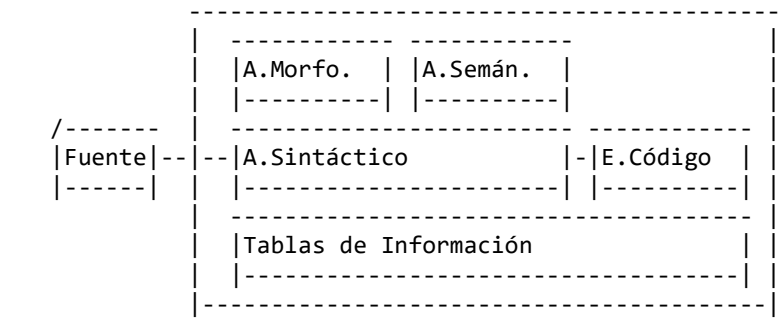


Pasos de compilación

En los compiladores de un solo paso suele fundirse el analizador semántico con el generador de código.

Partes de un intérprete

- Tablas de símbolos
- Análisis morfológico
- Análisis sintáctico
- Análisis semántico
- Ejecución de código
- Gestión de memoria
- Recuperación de errores



Analizador morfológico

Analizador lexical, preprocesador, "scanner".

Genera una unidad sintáctica o un vector de unidades sintácticas.

Realiza la primera fase de la reducción al axioma de la gramática.

Elimina espacios en blanco y comentarios.

Lenguaje regular.

Unidades sintácticas:

- Identificador
- Palabra reservada
- Constante numérica (entera, real...)
- Constante literal
- Símbolos simples
- Símbolos múltiples (+=, >=, ...)

Detección de errores morfológicos:

- Símbolo no permitido
- Identificador mal construido
- Constante numérica mal construida
- Constante literal mal construida
- Comentario mal construido

Analizador sintáctico

"Parser".

Gobierna todo el proceso.

Utiliza los demás como subrutinas.

Realiza el resto de la reducción al axioma de la gramática para comprobar que la instrucción en cuestión es correcta.

Lenguaje independiente del contexto.

Analizador semántico

Comprueba la corrección semántica de la instrucción.

Ejemplo: compatibilidad de tipo de las variables en una expresión.

Almacena información semántica en las tablas.

Generador de código

Traduce el programa fuente al lenguaje objeto.

Capítulo 2. Tablas de símbolos

Tabla: conjunto de pares clave-valor, llamados elementos de la tabla.

La tabla de símbolos es una componente necesaria de un compilador. Al declarar un identificador (normalmente una sola vez), éste es insertado en la tabla. Cada vez que se utilice el identificador se realizará una búsqueda en la tabla para obtener la información asociada (el valor).

Problemas asociados:

- Búsqueda: dada la clave de un elemento, encontrar su valor.
- Inserción: Dado un par clave-valor, añadir un elemento nuevo a la tabla.
- Cambio de valor: Buscar el elemento y cambiar su valor.
- Borrado: Eliminar un elemento de la tabla.

Longitud de búsqueda (o tiempo de acceso):

- De una clave: Li = número de comparaciones con elementos de la tabla para encontrar esa clave.
- Máxima: LM = número máximo de comparaciones para encontrar cualquier clave.
- Media (esperada): Lm = número medio de comparaciones para encontrar un valor.
  - Si la frecuencia de todas las claves es la misma:
  - Lm = (S Li)/N
  - Si la frecuencia de todas las claves no es la misma:
  - Lm = S pi.Li

Grado de ocupación:

```
s = n/N
```

donde n=número de elementos en la tabla y N=capacidad máxima de la tabla.

Función de búsqueda: B : K->E asocia a cada clave k un elemento B(k).

Valor asociado a una clave k: v(B(k)). Puede ser múltiple, en cuyo caso normalmente se convierte en un puntero. Si está en la tabla puede almacenarse consecutivamente o en subtablas paralelas.

Tablas de símbolos (identificadores)

La clave es el identificador. El valor está formado por:

- Atributos del identificador.
- Puntero a la posición de memoria asignada.

La clave puede sustituirse por un puntero.

Los identificadores pueden estar empaquetados.

La longitud del identificador puede especificarse en la tabla o delante del nombre, o ser implícita.

- Tablas consecutivas: Todos los elementos ocupan posiciones de memoria adyacentes.
- Tablas ligadas: cada elemento apunta al siguiente.
- Tablas doblemente ligadas: cada elemento apunta al siguiente y al anterior.

Tablas no ordenadas

Inserción: en el primer lugar vacío.

Búsqueda: secuencial, elemento a elemento.

```
Lm = (n+1)/2
LM = N
```

La búsqueda es muy lenta cuando el número de elementos es mayor que 20.

Tablas ordenadas

Los elementos se ordenan con algún criterio (p.e. alfabéticamente).

Búsqueda binaria o logarítmica.

Algoritmo de búsqueda en el bloque (1,n):

- Se mira el elemento (n+1)/2.
- Si es ese, encontrado.
- Si es menor, se busca en el bloque (1,(n-1)/2).
- Si es mayor, se busca en el bloque ((n+3)/2,n).

Longitud de búsqueda:

```
Lm = 1+log2 (n)
```

mucho mejor que en tablas no ordenadas.

Inserción:

- Se usa la búsqueda binaria para encontrar el elemento j tal que K(E(j))<k<K(E(j+1))
- Si la tabla es consecutiva, se corre toda la tabla un lugar desde E(j+1) hasta E(n) y se introduce el elemento nuevo en E(j+1).
- Si la tabla es ligada, se inserta un elemento nuevo entre E(j) y E(j+1).

En tablas consecutivas, si se van a insertar muchos elementos a la vez, podría convenir ordenarlos al final para evitar tantos traslados.

Tablas en árbol binario

Usan árboles binarios. Se compara la clave k con la del elemento. Si es mayor, se va a la derecha, si es menor a la izquierda. El tiempo de búsqueda depende del orden de inserción de los elementos y sólo es calculable si el árbol está equilibrado. Si no, se convierte en una lista ordenada, reordenando. Ej: G D M E A B F H. Secuencias aparentemente aleatorias pueden producir el mismo resultado.

Implementación: con dos punteros: izquierda, derecha; o con una tabla secuencial (2*i+1, 2*i+2).

Tablas de acceso directo

A cada clave se le asocia biunívocamente un elemento de la tabla mediante una función I(k) biyectiva.

Ejemplo: identificadores de una sola letra. La tabla tendrá 26/52 elementos como máximo.

```
I(k) = k-'A';  // (26 elementos)
I(k) = (k<'a') ? k-'A' : 26+k-'a'; // (52 elementos)
```

Búsqueda: dado k, se halla I(k) y se tiene el elemento.

```
Li = Lm = LM = 1
```

Problema: si hay muchos identificadores, I no podrá ser biyectiva. Llega un moment en que obtener I(k) lleva más tiempo que la búsqueda.

Tablas Hash o de entrada calculada

Es el método más usado. Se trata de trasformar la clave en un índice de entrada aplicándole una función Hash, I(k), que puede no ser biyectiva.

Es equivalente a una tabla de acceso directo mientras no aparezcan dos claves tales que I(k1) = I(k2): colisión.

Hay dos métodos principales para resolver la colisión.

Tablas Hash abiertas (con rehash)

Supongamos que los elementos de T son 0,1,...,N-1.

Búsqueda de la clave k.

- Se calcula h = I(k).
- Se compara k con T(h). Si es igual, encontrado.
- Si hay colisión (k!=T(h)&&T(h)!=NULL) se compara k con T(mod(h+p1,N)).
- Si hay nueva colisión se compara k con T(mod(h+p2,N)).
- ...
- Si hay nueva colisión se compara k con T(mod(h+pi,N)).

- hasta que se encuentre la clave buscada, un lugar vacío o se vuelva a T(h). En el primer caso, se ha encontrado. En el segundo, no está en la tabla. En el tercero, tampoco está, y la tabla está llena para ese valor de la función Hash.

Este almacenamiento se llama espaciado, porque los elementos ocupados están esparcidos por la tabla.

Tipos de rehash: lineal, aleatorio, multiplicativo, cuadrático.

Rehash lineal

pi = i

Se comparan elementos sucesivos. Problema: apiñamiento de elementos.

La longitud de búsqueda es difícil de calcular (depende del grado de apiñamiento, y éste del orden en que se definen los identificadores).

Peterson realizó simulaciones. Schay y Spruth aplicaron la hipótesis de que en el apiñamiento todos los elementos de la posición i aparecen antes que los de la posición i+1 y obtuvieron la fórmula:

```
Lm = (1-s/2)/(1-s)
```

Resultados:

s	Peterson	Schay/Spruth	A.C.M.
0.1	1.053	1.056	
0.2	1.137	1.125	1.43
0.3	1.230	1.214	
0.4	1.366	1.333	2.35
0.5	1.541	1.500	
0.6	1.823	1.750	3.24
0.7	2.260	2.167	
0.8	3.223	3.000	5.22
0.9	5.526	5.500	
1.0	16.914	¥	14.67

Es mucho mejor que la búsqueda binaria. No depende del número de elementos, sino del grado de ocupación.

Rehash aleatorio

pi = pseudoaleatorio.

Se elimina el apiñamiento.

```
Lm = -loge(1-s)/s
```

Resultados:

s	Lm
0.1	1.05

```
0.5       1.39
0.9       2.56
```

Rehash multiplicativo

pi = ih, donde h es el índice original.

Se prueban los elementos h, 2h, 3h, ... (mod N)

Funciona bien si N es primo (cubre toda la tabla).

h nunca puede valer 0. La tabla irá de 1 a N-1.

Rehash cuadrático

pi = a.i*2+b.i+c

Hay que procurar que cubra lo más posible la tabla.

Si N = 2*p no cubre casi.

Si N es primo, cualquier combinación de a,b,c cubre media tabla. Es peor que el aleatorio, pero Lm y el tiempo de cálculo de pi son mejores.

Ejemplo: sea c=0, para incluir p0 (h=h+p0 => p0=0 => c=0).

```
dpi = p(i+1)-p(i) = 2.a.i+a+b
```

Hacemos a=-1/2 para que el coeficiente de i sea -1.

dpi disminuye una unidad a medida que i crece. Podemos tener una variable con el valor del incremento e ir disminuyendo una unidad cada vez.

```
dpi = -i+b-1/2
```

Se puede usar esa variable para parar. Por ejemplo, si la tabla tiene 787 elementos, tendremos que pararnos tras 393 intentos. Si hacemos dp0=392, hay que parar cuando dpi valga -1. Entonces b=392+1/2. Y la fórmula queda:

```
pi = -i*2/2+(392+1/2)i
pi = p(i-1) + dp(i-1)
p0 = 0
dp0 = 392
dpi = dp(i-1) - 1
```

Tablas Hash con encadenamiento

Puede ser interno o con "overflow"

Tabla Hash con encadenamiento interno

- Vector Hash VH: contiene punteros a la tabla para el primer elemento que corresponde a cada valor de la función Hash.
- Tabla propiamente dicha: contiene tríadas (clave, valor, encadenamiento).
- Inserciones secuenciales. Hay un puntero a la última posición ocupada o la primera libre.

Proceso de búsqueda:

1. Se calcula H(k).
2. Se indexa VH(H(k))=t.
3. Si t es NULL no está el elemento. Se sale.
4. Se compara k con K(t). Si son iguales, encontrado. Se sale.
5. Se hace t=E(t) y volvemos al paso 3.

Proceso de inserción. Sea L el puntero a la primera posición libre de la tabla:

1. Se calcula H(k).
2. Se indexa VH(H(k))=t.
3. Si t es NULL no está el elemento. Si la tabla no está llena, se hace *L=(k,v,NULL); H(k)=L; L++ y se sale.
4. Se compara k con K(t). Si son iguales, error. Se sale.
5. Se hace t=E(t).
6. Si t no es NULL se vuelve al paso 4.
7. Si la tabla no está llena, se hace *L=(k,v,NULL); E(t)=L; L++ y se sale.

Si la tabla se llena, se puede alargar mediante la memoria dinámica. También se puede emplear una tabla ligada, en vez de secuencial.

El vector Hash debe estar inicializado a NULL.

```
Lm = 1 + (n-1) / (2N)
s=1 => Lm = 1 + (N-1)/(2N) < 1.5
```

Supresión de entradas: o bien se pone marca de entrada suprimida o un doble encadenamiento. En el primer caso, la inserción no se hace en la primera posición vacía, sino en la primera posición suprimida por la que se pasó, quitándole la marca.

Encadenamiento con "overflow"

Es igual, sólo que el vector Hash se amplía para incluir también (k,v) y se incorpora a la tabla. Se reduce ligeramente el tiempo de búsqueda.

Función Hash

1. Suma/O exclusivo de las letras del identificador, formando una palabra (un byte) W.
2. Se reduce W a un índice para el vector Hash.

1. Si la tabla tiene N = 2*p entradas, se cogen los p bits medios de WxW.
2. O se usa una función lógica (ej. O exclusivo) con partes de W.
3. O se descompone W en secciones de p bits, se suman y nos quedamos con los p bits de la derecha.
4. O se usa el resto de W/N.

   En PL/I IBM se usa esta versión con 211 elementos en el vector Hash.

5. O se toman los p últimos bits.

Tablas de intérpretes APL.

Tablas de símbolos de estructura de bloques.

Utiles para lenguajes de estructura de bloques: Algol, PL/I, Pascal, C.

Ejemplo:

```
{                   // Bloque 1
  int a, b, c, d;
  {                 // Bloque 2
    int e, f;
    L1:
  }
  {                 // Bloque 3
    int g, h;
    L2: {           // Bloque 4
          int a;
        }
    L3:
  }
}
```

Cualquier línea de un programa está contenida en uno o más bloques que definen ámbitos de validez de nombres. El ámbito definido por el bloque más profundo que contiene la instrucción que estamos analizando, se llama ámbito actual. Los ámbitos que rodean a una línea de un programa son abiertos respecto a esa línea. Los que no la rodean son cerrados respecto a esa línea. En el ejemplo anterior, si tomamos como referencia la línea "int a;", los ámbitos abiertos son los correspondientes a los bloques 1, 3 y 4. El bloque 2 es un ámbito cerrado. El ámbito actual es el bloque 4.

Para saber qué nombres están activos en un punto de un programa, se utilizan las siguientes reglas:

- Sólo son accesibles los nombres definidos en el ámbito actual y en los ámbitos abiertos que le rodean.
- Si un nombre se declara en más de un ámbito abierto, sólo es accesible el que está definido en el ámbito más profundo.

Son inaccesibles las variables definidas en un ámbito cerrado. En el ejemplo anterior, los nombres e, f y L1: son inaccesibles desde los bloques 3 y 4. Desde el bloque 4 es accesible el "int a" definido en el propio bloque, pero no "int a", definido en el bloque 1.

Los nombres de los argumentos de un programa son locales al programa. El nombre del programa es local al bloque donde se declaró el programa y global al propio programa. Si no fuera así, el programa sería inaccesible.

Hay dos modos de implantar las tablas de símbolos de bloques:

- Una tabla por ámbito
- Una tabla única para todos los ámbitos.

La tabla única suele utilizarse en compiladores de un solo paso, en los que se puede descartar la información referente a un ámbito en cuanto se cierra. Un compilador de múltiples pasos suele usar una tabla individual por ámbito.

Una tabla por ámbito

Hay que asegurarse de que sólo son accesibles los nombres que cumplen las reglas. Para ello se crea una lista de ámbitos. Los ámbitos abiertos se encuentran situados en la lista en orden de apertura, con el actual en primer lugar. El problema surge con el manejo de los ámbitos cerrados. En este caso hay que tener en cuenta el tipo de compilador:

- Si el compilador es de un solo paso, los ámbitos cerrados se pueden descartar, y la lista se convierte en una pila. Para buscar un nombre se empieza por el ámbito situado en lo alto de la pila y se va descendiendo hasta que se encuentra o hasta que se acaba la pila.

  En el ejemplo anterior, la pila de ámbitos tendría la siguiente estructura a lo largo del análisis:

```
                      Bloque 2: e,f,L1:     Bloque 3: i,h,L2:,L3:
  Bloque 1: a,b,c,d     Bloque 1: a,b,c,d     Bloque 1: a,b,c,d

  Bloque 4: a
  Bloque 3: i,h,L2:,L3: Bloque 3: i,h,L2:,L3:
  Bloque 1: a,b,c,d     Bloque 1: a,b,c,d     Bloque 1: a,b,c,d
```

- Si el compilador tiene más de un paso, hará falta guardar la información relativa a los ámbitos cerrados para utilizarla en pasos posteriores. En este caso, la lista de ámbitos del ejemplo tendría la siguiente estructura a lo largo del análisis:

```
                                        Bloque 2: e,f,L1:
                      *Bloque 2: e,f,L1:    *Bloque 3: i,h,L2:,L3:
   *Bloque 1: a,b,c,d     Bloque 1: a,b,c,d     Bloque 1: a,b,c,d

    Bloque 2: e,f,L1:     Bloque 2: e,f,L1:     Bloque 2: e,f,L1:
```

- *Bloque 4: a Bloque 4: a Bloque 4: a
- Bloque 3: i,h,L2:,L3:*Bloque 3: i,h,L2:,L3: Bloque 3: i,h,L2:,L3:
- Bloque 1: a,b,c,d Bloque 1: a,b,c,d *Bloque 1: a,b,c,d

donde son activos el bloque marcado con un asterisco y los situados por debajo del mismo. Los bloques nuevos se insertan en la tabla al analizar el símbolo {. Cuando aparece el símbolo }, el bloque queda cerrado y los nuevos bloques se insertan por debajo de él.

La búsqueda se realiza sólo en el bloque actual si se trata de una declaración, o en él y sus antepasados en cualquier otro caso.

Los problemas que presenta esta implantación son:

- Búsqueda en múltiples tablas de símbolos, con la consiguiente pérdida de tiempo.
- Fragmentación del espacio de almacenamiento de las tablas

Una sola tabla de símbolos

En este caso, todos los identificadores de todos los ámbitos están en la misma tabla. Cada identificador lleva información sobre el ámbito al que pertenece. Un identificador puede aparecer varias veces, siempre que cada aparición lleve un número de ámbito diferente. En el ejemplo anterior:

Bloque	Bloque superior	Nº elementos	Puntero
1	0	4	4
2	1	3	1
3	1	4	3
4	3	1	2

Subtablas, guardadas por orden de aparición de }

```
1:   e, f, L1
2:   a
3:   g, h, L2, L3
4:   a, b, c, d
```

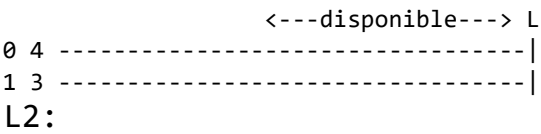
Procedimiento de creación:

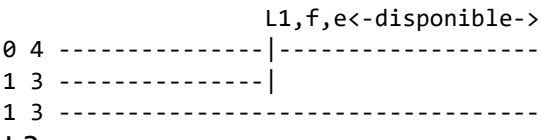
El final de la tabla de símbolos se usa como una lista al revés. Cuando aparece { se añade un bloque y se pone el puntero al final de la lista. Cuando llega } se vacía la lista correspondiente al bloque que terminó, copiando al principio de la tabla.

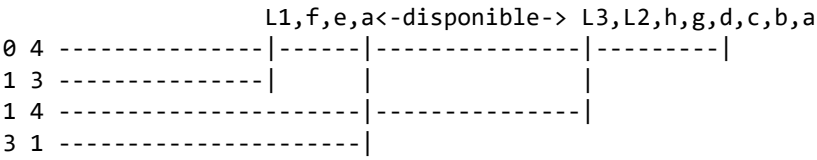
Inserción: se añade a la lista del final de la tabla de símbolos, se corre el puntero y se aumenta en 1 el número de elementos de ese bloque.

Búsqueda: se busca en el bloque presente sólo (si es una declaración) o en él y sus antepasados en otro caso.

Ejemplo:

- En L1:
- 

- En L2:
- 

- En L3:
- 

Las ventajas de esta implantación respecto a la de múltiples tablas son:

- La búsqueda es más rápida, ya que sólo hay que buscar en una tabla.
- El espacio ocupado es menor, pero la diferencia queda compensada con el hecho de que en esta implantación hay un campo más (ámbito).

El valor

Puede depender del identificador (tamaño variable). Si no, pueden ponerse varios tamaños estándar o poner un puntero al valor.

Información requerida:

- Clase de identificador: variable, función/procedimiento, etiqueta, tipo, valor de enumeración, etc.
- Tipo: entero, real, lógico, complejo, carácter, cadena, estructura, tipo declarado, etc.; función con resultado, sin resultado, operador, cuántos argumentos, etc.
- Precisión, escala
- Forma: escalar, "array", lista, etc.
- Dimensiones: número, valores si son constantes.
- Dirección asociada.
- Si se ha declarado.
- Si es un parámetro de una rutina, de un FOR, etc.
- Si se le ha asignado valor.
- Si tiene valor inicial.
- Si está en un COMMON, EQUIVALENCE, etc.
- Si es estático, dinámico, automático, externo, etc.
- Si es función recursiva.
- Lista de parámetros, variables locales, etc.

Esta información puede agruparse en bits empaquetados.

Capítulo 3. Análisis morfológico

Analizador lexical, preprocesador, "scanner".

Genera una unidad sintáctica o un vector de unidades sintácticas.

Realiza la primera fase de la reducción al axioma de la gramática.

Elimina espacios en blanco y comentarios.

Lenguaje regular.

Unidades sintácticas:

- Identificador
- Palabra reservada
- Constante numérica (entera, real...)
- Constante literal
- Símbolos simples
- Símbolos múltiples (+=, >=, ...)

Gramática de un analizador morfológico

Axioma: <US> (Unidad Sintáctica).

```
<US> ::= <Id> | <PR> | <Numero> | <Literal> | <Caracter> |
         <SS> | <SM>
<Id> ::= <Letra> | <Letra> <IdC>
<IdC> ::= <Letra> <IdC> | <Cifra> <IdC> | <Letra> | <Cifra>
<Numero> ::= <Entero> | <Real>
<Entero> ::= <EnteroSS> | - <EnteroSS>
<EnteroSS> ::= <Cifra> | <Cifra> <EnteroSS>
<Real> ::= <PFijo> | <PFijo> <Exponente>
<PFijo> ::= <Entero> . <EnteroSS> | . <EnteroSS> | <Entero> .
<Exponente> ::= E <Entero>
<Literal> ::= "" | " <Cadena> "
<Cadena> ::= <Simbolo> | <Simbolo> <Cadena>
<Caracter> ::= ' <Simbolo> '
<SS> ::= + | - | * | / | = | < | > | ( | )
<SM> ::= =+ | != | <= | >= | ++ | --
<Simbolo> ::= <Letra> | <Cifra> | <SS> | ! | . | , | b |
              \' | \" | \n
<Letra> ::= A | B | ... | Z | a | b | ... | z
<Cifra> ::= 0 | 1 | ... | 9
```

Eliminación de espacios

Puede hacerse mediante una rutina semántica o mediante reglas sintácticas:

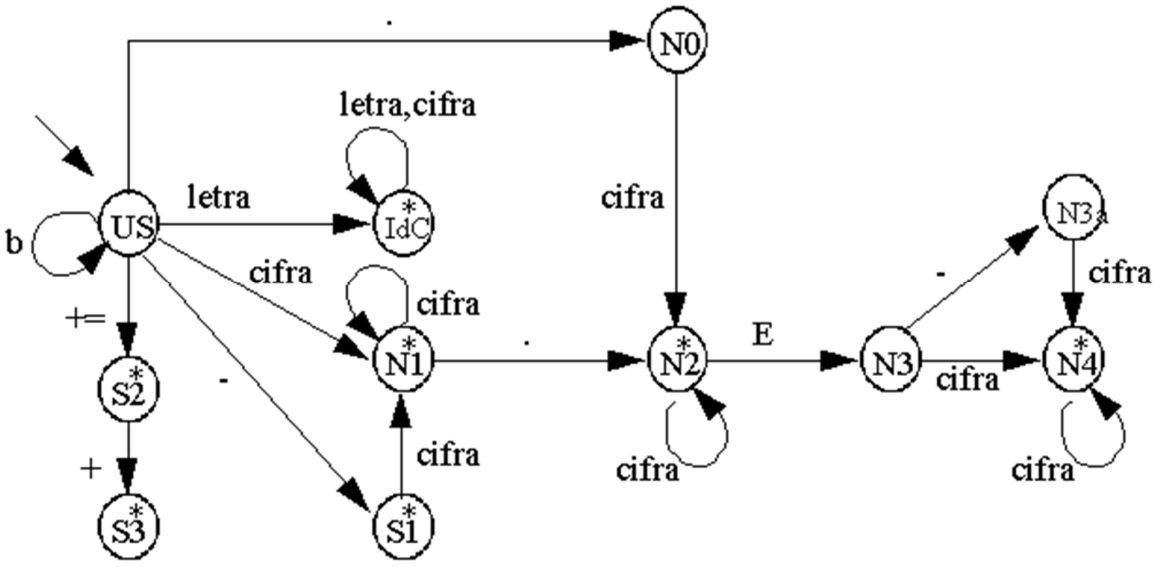
```
<US> ::= b <US>
<IdC> ::= b <IdC>
etc.
```

Eliminación de comentarios

Puede hacerse mediante una rutina semántica o mediante reglas sintácticas:

```
<Comentario> ::= /**/ | /* <Cadena> */
```

Autómata que realiza un analizador morfológico



Añadir análisis de comentarios.

Extraer la gramática tipo 3 del autómata.

Semántica del analizador morfológico

A cada regla de la gramática se le asocia una acción semántica, si hace falta.

Ejemplos:

```
<Id> ::= f0 <Letra> f1 f2 <IdC> f3
<IdC> ::= <Letra> f1 f2 <IdC> | <Cifra> f1 f2 <IdC> | ^
```

donde

- f0: inicializar un contador.
- f1: sumar 1 al contador.
- f2: copiar el carácter a la primera posición de un "buffer".
- f3: marcar el fin de la cadena en el "buffer". Si el contador es mayor que el tamaño máximo, error. Comprobar si el identificador es palabra reservada comparando con una tabla. Si no lo es, insertar en tabla de símbolos y devolver un puntero a la tabla.

```
<Entero> ::= g0 <EnteroSS> | g0 - <EnteroSS> g2
<EnteroSS> ::= <Cifra> g1 | <Cifra> g1 <EnteroSS>
```

donde

- g0: inicializar una variable entera N:=0

- g1: N := (10*N) + cifra.
- g2: N = -N

Resultado: N.

Funciones semánticas típicas:

- Avanzar un símbolo en la cadena de entrada.
- Cálculo de un identificador.
- Cálculo de un número.
- Separación de una cadena de caracteres.
- Fusión de símbolos dobles en un solo símbolo sintáctico.

Construir un programa en C.

Detección de errores morfológicos

- Símbolo no permitido
- Identificador demasiado largo
- Identificador mal construido
- Constante numérica demasiado larga
- Constante numérica mal construida
- Constante literal mal construida
- Comentario mal construido

Capítulo 4. Análisis sintáctico

Lenguaje independiente del contexto.

"Parser". Gobierna todo el proceso. Realiza el resto de la reducción al axioma de la gramática para comprobar que la instrucción en cuestión es correcta. Considera como símbolos terminales las unidades sintácticas devueltas por el analizador morfológico.

Dos tipos principales de análisis:

- De arriba a abajo ("top-down").
- De abajo a arriba ("bottom-up").

Que sea una gramática de tipo 2 (lenguaje independiente del contexto) no nos asegura que el autómata a pila sea determinista. Estos sólo representan un subconjunto de los lenguajes i.c.. A lo largo de las páginas siguientes iremos imponiendo diversas restricciones a las gramáticas. Las restricciones de un método no tienen porqué ser las mismas que las de otro (complementariedad).

Análisis "top-down"

Se parte del axioma y se va realizando la reducción. La palabra x (normalmente una instrucción) se llama meta u objetivo del análisis. La primera fase del análisis consiste en encontrar, entre las reglas cuya parte izquierda es el axioma, la que conduce a x.

Análisis "top-down" con vuelta atrás lenta

Sea el axioma S y las reglas cuya parte izquierda es el axioma:

```
S ::= X1 X2 ... Xn | Y1 Y2 ... Ym | ...
```

Sea x la palabra a reconocer. El objetivo del análisis es encontrar una derivación tal que

```
S ->* x
```

Probaremos primero la regla

```
S ::= X1 X2 ... Xn
```

Debemos descomponer x en la forma

```
x = x1 x2 ... xn
```

y tratar de encontrar las derivaciones

```
X1 ->* x1
X2 ->* x2
   ...
Xn ->* xn
```

Cada una de las derivaciones anteriores es una submeta. Pueden ocurrir los siguientes casos:

- Xi = xi: submeta reconocida. Pasamos a la submeta siguiente.
- Xi != xi y Xi en AT*: submeta rechazada. Intentamos encontrar otra submeta válida en X(i-1). Si i=1, elegimos la siguiente parte derecha para el mismo símbolo no terminal a cuya parte derecha pertenece Xi. Si es el último, elegimos la siguiente parte derecha del símbolo no terminal anterior. Si éste es el axioma, la cadena x queda rechazada.
- Xi es un símbolo no terminal. Buscamos las reglas de las que Xi es parte izquierda:
- ```
  Xi ::= Xi1 Xi2 ... Xin | Yi1 Yi2 ... Yim | ...
  ```

  elegimos la primera opción:

  ```
  Xi ::= Xi1 Xi2 ... Xin
  ```

  descomponemos xi en la forma

  ```
  xi = xi1 xi2 ... xin
  ```

  lo que nos da las nuevas submetas

  ```
  Xi1 ->* xi1
  ```
  ```

```
Xi2 ->* xi2
   ...
Xin ->* xin
```

y continuamos recursivamente igual que antes.

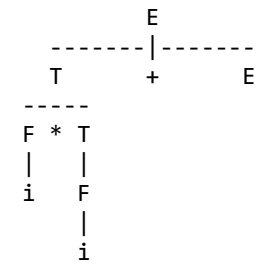
Análisis "top-down" con vuelta atrás rápida

Consiste en ordenar las distintas partes derechas de las reglas que comparten la misma parte izquierda de manera que la regla "buena" se analice antes.

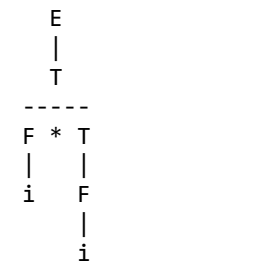
Ejemplo:

```
E ::= T+E | T
T ::= F*T | F
F ::= i
```

Analizamos la cadena i*i. Probamos primero E::=T+E y obtenemos el árbol



En cuanto comprobemos que el signo + no pertenece a la cadena objetivo, no es necesario volver a las fases precedentes del análisis tratando de obtener otra alternativa, sino que pasamos directamente a intentar E::=T.



Una buena regla para ordenar la gramática es poner primero las partes derechas más largas. Si una parte derecha es la cadena vacía, debe ser la última. Si se llega a ella, automáticamente hay éxito en la submeta.

Forma normal de Greibach

Ver "Teoría de Lenguajes, gramáticas y autómatas".

Lenguajes LL(1)

Son gramáticas en forma normal de Greibach en las que no existen dos reglas con la misma parte izquierda, cuya parte derecha empiece por el mismo símbolo terminal.

Conversión: si existen reglas de la forma:

```
U ::= aV | aW
V ::= bX | cY
W ::= dZ | eT
```

tratamos de sustituir las dos reglas de parte izquierda U por

```
U ::= aK
K ::= bX | cY | dZ | eT
```

y aplicamos la misma sustitución a las nuevas reglas formadas.

La gramática se llama LL(1) de Knuth porque basta estudiar en cada momento un carácter de la cadena objetivo para saber qué regla se debe aplicar.

Análisis "top-down" selectivo

Se llama también "sin vuelta atrás" o "descenso recursivo". Se basa en poner la gramática en la forma normal de Greibach (LL(1)) y marchar directamente por el único camino permisible. Si en algún momento no se encuentra regla que aplicar, se puede generar directamente un mensaje de error.

Ejemplo: sea la gramática

```
E := T + E | T - E | T
T := F * T | F / T | F
F ::= i | (E)
```

En forma normal de Greibach queda:

```
E ::= iPTME | (ECPTME | iDTME | (ECDTME | iME | (ECME |
      iPTSE | (ECPTSE | iDTSE | (ECDTSE | iSE | (ECSE |
      iPT   | (ECPT   | iDT   | (ECDT   | i   | (EC
T ::= iPT   | (ECPT   | iDT   | (ECDT   | i   | (EC
F ::= i | (EC
M ::= +
S ::= -
P ::= *
D ::= /
C ::= )
```

En forma LL(1) queda:

```
E ::= iV  | (ECV
V ::= *TX | /TX  | +E | -E | ^
X ::= +E  | -E   | ^
T ::= iU  | (ECU
U ::= *T  | /T   | ^
F ::= i   | (EC
C ::= )
```

Paso automático de la gramática LL(1) al analizador

Dada una gramática LL(1), formada por un conjunto de reglas de la forma

```
U ::= x X1 X2 ... Xn | y Y1 Y2 ... Ym | ... | z Z1 Z2 ... Zp
```

Generamos la siguiente función C:

```
int U (char *cadena, int i)
{
  if (i<0) return i;  /* Pasa errores anteriores */
  switch (cadena[i]) {
    case x:
      i++;
      i = X1 (cadena, i);
      i = X2 (cadena, i);
      . . .
      i = Xn (cadena, i);
      break;
    case y:
      i++;
      i = Y1 (cadena, i);
      i = Y2 (cadena, i);
      . . .
      i = Ym (cadena, i);
      break;
    ...
    case z:
      i++;
      i = Z1 (cadena, i);
      i = Z2 (cadena, i);
      . . .
      i = Zp (cadena, i);
      break;
    /* Fin de cadena va a default */
    default: return -n; /* Genera error n */
      }
  return i;
}
```

Si las reglas tienen la forma

```
U ::= x X1 X2 ... Xn | ... | z Z1 Z2 ... Zp | ^
```

Generamos la siguiente función C:

```
int U (char *cadena, int i)
{
  if (i<0) return i;  /* Pasa errores anteriores */
  switch (cadena[i]) {
    case x:
      i++;
      i = X1 (cadena, i);
      i = X2 (cadena, i);
      . . .
      i = Xn (cadena, i);
      break;
    ...
    case z:
      i++;
      i = Z1 (cadena, i);
      i = Z2 (cadena, i);
      . . .
      i = Zp (cadena, i);
      break;
      }
```

```
  return i;
}
```

Ejemplo: la gramática anterior se reduce a:

```
int E (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case 'i':
      i++;
      i = V (cadena, i);
      break;
    case '(':
      i++;
      i = E (cadena, i);
      i = C (cadena, i);
      i = V (cadena, i);
      break;
    default: return -1;
      }
  return i;
}

int V (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case '*':
    case '/':
      i++;
      i = T (cadena, i);
      i = X (cadena, i);
      break;
    case '+':
    case '-':
      i++;
      i = E (cadena, i);
      break;
      }
  return i;
}

int X (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case '+':
    case '-':
      i++;
      i = E (cadena, i);
      break;
      }
  return i;
}

int T (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case 'i':
      i++;
      i = U (cadena, i);
      break;
    case '(':
      i++;
```

```
      i = E (cadena, i);
      i = C (cadena, i);
      i = U (cadena, i);
      break;
    default: return -2;
      }
  return i;
}

int U (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case '*':
    case '/':
      i++;
      i = T (cadena, i);
      break;
      }
  return i;
}

int F (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case 'i':
      i++;
      break;
    case '(':
      i++;
      i = E (cadena, i);
      i = C (cadena, i);
      break;
    default: return -3;
      }
  return i;
}

int C (char *cadena, int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case ')':
      i++;
      break;
    default: return -4;
      }
  return i;
}
```

La cadena x se analiza invocando

```
axioma (x, 0);
```

En nuestro ejemplo:

```
E (x, 0);
```

Si devuelve strlen(x), la cadena x es correcta. En caso contrario, devolverá un número negativo que indica el error detectado.

Ejemplo: análisis de "i+i*i".

```
E ("i+i*i", 0) = V ("i+i*i", 1) =
               = E ("i+i*i", 2) =
               = V ("i+i*i", 3) =
               = X ("i+i*i", T ("i+i*i", 4)) =
               = X ("i+i*i", U ("i+i*i", 5)) =
               = X ("i+i*i", 5) =
               = 5
```

Ejemplo: análisis de "i+i*".

```
E ("i+i*", 0) = V ("i+i*", 1) =
              = E ("i+i*", 2) =
              = V ("i+i*", 3) =
              = X ("i+i*", T ("i+i*", 4)) =
              = -2
```

Análisis "bottom-up"

Parte del objetivo y trata de reconstruir las producciones para llegar a reducir al axioma.

Ejemplo: sea la gramática

```
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= i | (E)
```

y la expresión (reducida por el analizador morfológico) i+i*i.

Colocamos un puntero sobre la cadena a analizar.

```
meta: E

i + i * i
^
```

La única regla que empieza por "i" es F::=i. La aplicamos:

```
meta: E

F
|
i + i * i
^
```

La cadena a obtener (subobjetivo), vista desde arriba, es F+i*i. La única regla aplicable (que empiece por F) es T::=F. La aplicamos:

```
meta: E

T
|
F
|
i + i * i
^
```

La cadena a obtener (subobjetivo), vista desde arriba, es T+i*i. Hay dos reglas cuya parte derecha empieza por T, E::=T y T::=T*F. Probamos la de

parte derecha más larga (la segunda): regla aplicable (que empiece por F) es T::=F. La aplicamos:

```
meta: E

T
|----
T * F
|
F
|
i + i * i
  ^
```

y avanzamos el puntero para comprobar. La submeta a encontrar es *F y la subcadena a generar es +i*i. Como comienzan por distinto símbolo terminal, no es posible hacerlo. Retrocedemos a la situación anterior y aplicamos la otra regla, E::=T.

```
meta: E

E
|
T
|
F
|
i + i * i
^
```

El objetivo actual es E+i*i. Ha aparecido el axioma, pero queda cadena por reducir. Sólo hay una regla cuya parte derecha empiece por E, E::=E+T. La aplicamos:

```
meta: E

E
|----
E + T
|
T
|
F
|
i + i * i
  ^
```

Ahora tenemos el subobjetivo +T que debe producir +i*i. Los dos símbolos terminales iniciales coinciden. Los identificamos y avanzamos el puntero.

```
meta: E

E
|----
E | T
| | Submeta: T
T |
| |
F |
| |
i + i * i
    ^
```

Partiendo de i*i hay que comprobar la submeta T. La única regla que empieza por "i" es F::=i. La aplicamos:

```
meta: E

E
|----
E | T
| | Submeta: T
T |
| |
F | F
| | |
i + i * i
    ^
```

La única regla que empieza por F es T::=F. La aplicamos:

```
meta: E

E
|----
E | T
| | Submeta: T
T | T
| | |
F | F
| | |
i + i * i
    ^
```

Hemos encontrado la submeta T. Pero aún queda cadena. Miramos si hay alguna regla cuya parte derecha empiece por T. Hay dos, E::=T y T::=T*F. Probamos la de parte derecha más larga (la segunda).

```
meta: E

E
|----
E | T
| | Submeta: T
| | T
| | |----
T | T * F
| | |
F | F
| | |
i + i * i
      ^
```

Avanzamos el puntero y comprobamos la subcadena *F con la subcadena *i. Los dos símbolos terminales iniciales coinciden. Los identificamos y avanzamos:

```
meta: E

E
|----
E | T
| | Submeta: T
| | T
| | |----
```

```
T | T | F
| | | | submeta: F
F | F |
| | | |
i + i * i
        ^
```

Partiendo de i hay que comprobar la submeta F. La única regla que empieza por "i" es F::=i. La aplicamos:

```
meta: E

E
|----
E | T
| | Submeta: T
| | T
| | |----
T | T | F
| | | | Submeta: F
F | F | F
| | | | |
i + i * i
        ^
```

Todas las metas y submetas coinciden. Luego ésta es la reducción al axioma buscada. El árbol de análisis queda:

```
E
|----
E | T
| | |----
T | T | F
| | | | |
F | F | |
| | | | |
i + i * i
```

Gramáticas de precedencia simple

Introducir aquí la teoría de formas sentenciales, frases y asideros, y la de relaciones.

Relaciones de precedencia

Dada una gramática limpia G de axioma Z, definimos las siguientes relaciones de precedencia entre los símbolos del vocabulario A = AN U AT. Para todo R,S en A:

```
R =. S <=> existe U::=uRSv  en  P
R <. S <=> existe U::=uRTv  en  P,
           tal que T F+ S en P.
R >. S <=> existe U::=uTWv  en  P,
           tal que T L+ R, W F* S en P.
R <.= S <=> R <. S ó R =. S
R >.= S <=> R >. S ó R =. S
```

Teorema: *R =. S <=> RS aparece en el asidero de alguna forma sentencial*.

Prueba:

- =>
  - R =. S => existe U::=uRSv en P
  - G es limpia => Z ->* xUy => existe un árbol que produce xUy
  - Podamos ese árbol hasta que U esté en un asidero (todo símbolo ha de ser alguna vez parte de un asidero).
  - Aplicamos la regla U::=uRSy. Como U era parte del asidero, el asidero de este árbol nuevo debe ser uRSv, q.e.d.
- <=

  RS es parte de un asidero. Por definición de asidero existe una regla U::=uRSy => R =. S, q.e.d.

Teorema: *R >. S <=> existe una forma sentencial uRSv donde R es el símbolo final de un asidero*.

Prueba:

- =>
  - R >. S => existe U::=uTWv en P, T L+ R, W F* S
  - G es limpia => Z ->* xUy ->+ xuTWvy
  - T L+ R => T ->+ tR => Z ->+ xurRWvy
  - Dibujamos este árbol y reducimos hasta que R sea parte del asidero. Por construcción, tendrá que ser su último símbolo.
  - W F* S => W ->* Sw => Z ->+ xurRSwvy
  - Añadimos al árbol anterior esta última derivación. El asidero no ha cambiado, R sigue siendo el último símbolo y va seguido por S, q.e.d.
- <=

  R es la cola de un asidero y va seguido por S. Reducimos hasta que S esté en el asidero. Esto da un árbol para la forma sentencial xTSy, donde T ->* tR.

  - Si T y S están los dos en el asidero, existe U ::= uTSv => R >. S q.e.d.
  - Si S es cabeza del asidero, reducimos hasta llegar a xTWw donde T está en el asidero. Ahora, W ->+ S y W tiene que estar en el asidero, pues si no está T sería la cola y habría sido asidero antes que S. Luego R >. S q.e.d.

Teorema: *R <. S <=> existe una forma sentencial uRSv donde S es la cabeza de un asidero*.

Se demuestra de forma análoga.

Gramática de precedencia simple

Definición: *G es una G.P.S. o G.P.(1,1) si:*

1. *Sólo existe, como mucho, una relación de precedencia entre dos símbolos cualesquiera del vocabulario.*
2. *No existen dos producciones con la misma parte derecha.*

Se llama G.P.(1,1) porque sólo se usa un símbolo a cada lado de un posible asidero para decidir si lo es o no.

Problema: si no se cumple esto, se puede manipular la gramática para intentar que se cumpla. La recursividad a izquierdas o a derechas da problemas (salen dos relaciones de precedencia con los símbolos que van antes o después del recursivo). Para evitarlo, se puede estratificar la gramática, añadiendo símbolos nuevos, pero si hay muchas reglas es muy pesado.

Supondremos que toda forma sentencial queda rodeada por dos símbolos especiales de principio y fin de cadena: |- -|, que no están en A y tales que, para todo S en A, |- <. S y S >. -|.

Teorema: *Una gramática de precedencia simple no es ambigua. Además, el asidero de una forma sentencial S1 ... Sn es la subcadena Si ... Sj, situada más a la izquierda tal que*:

```
Si-1 <. Si =. Si+1 =. ... =. Sj-1 =. Sj >. Sj+1
```

Prueba:

- Si Si ... Sj es asidero, se cumple la relación por los teoremas anteriores.
- Reducción al absurdo. Se cumple la relación y no es asidero. Entonces ninguna poda la hará asidero, pues si en algún momento posterior resultara ser la rama completa más a la izquierda, ya lo es ahora.

  Si no es asidero, cada uno de los símbolos de Si-1 Si ... Sj Sj+1 debe aparecer más pronto o más tarde como parte de un asidero. Sea St el primero que aparece.

  1. Si St=Si-1, de los teoremas se sigue que Si-1 =. Si ó Si-1 >. Si, lo que contradice que Si-1 <. Si (no puede haber dos relaciones entre dos símbolos).
  2. Si St=Sj+1, de los teoremas se sigue que Sj =. Sj+1 ó Sj <. Sj+1, lo que contradice que Sj >. Sj+1.
  3. Si i-1<t<j+1, ocurre lo siguiente:
     1. Que Si-1 no puede estar en el asidero (pues entonces Si-1 =. Si, contradicción).
     2. Que Sj+1 no puede estar en el asidero (pues entonces Sj =. Sj+1, contradicción).
     3. Que ningún símbolo Sk, i<k<=t puede ser cabeza del asidero (pues entonces Sk-1 <. Sk, contradicción).

4. Que ningún símbolo Sk, t<=k<j puede ser cola del asidero (pues entonces Sk <. Sk+1, contradicción).
5. Luego la cabeza del asidero debe ser Si y la cola Sj, lo que contradice que Si ... Sj no era el asidero, q.e.d.

Como el asidero es único (por construcción) y sólo puede ser parte derecha de una regla (por ser G.P.S.) sólo se puede aplicar una regla para reducir. Luego la gramática no es ambigua.

Construcción de las relaciones

1. La relación =. se construye por simple observación de las partes derechas de las reglas (definición de =.).
2. La relación <. se construye así:
3. <. = (=.) +.x (F+)

   donde +.x es el producto booleano de matrices. (Por definición del producto de relaciones y la definición de <. y =.).

4. La relación >. se construye así:
5. >. = (L+)' +.x (=.) +.x (F*)

   donde ' es la transposición de matrices. La demostración queda como ejercicio.

Algoritmo de análisis

1. Se almacenan las producciones de la gramática en una tabla.
2. Se construye la matriz de precedencia MP de dimensiones N.N (N=cardinal de A), tal que
3. MP(i,j) = 0 si no existe relación entre Si y Sj
4. = 1 si Si <. Sj
5. = 2 si Si =. Sj
6. = 3 si Si >. Sj
7. Se inicializa una pila con el símbolo |- y se añade el símbolo -| al final de la cadena de entrada.
8. Se compara el símbolo de lo alto de la pila con el siguiente símbolo de entrada.
9. Si no hay relación, error sintáctico: cadena rechazada.
10. Si hay la relación <. o la relación =., se introduce el símbolo de entrada en la pila y se avanza el puntero sobre la entrada. Volver al paso 4.
11. Si la relación es >., el asidero termina en lo alto de la pila.
12. Se recupera el asidero de la pila, sacando símbolos hasta que el símbolo en lo alto de la pila está en relación <. con el último sacado.
13. Se compara el asidero con las partes derechas de las reglas.
14. Si no está, error sintáctico: cadena rechazada.
15. Si está, se introduce la parte izquierda de la regla en la pila.

16. Si en la pila sólo queda |- seguido del axioma y la cadena de entrada ha quedado reducida al símbolo -|, la cadena ha sido reconocida.
17. Volver al paso 4.

Ejemplo:

```
S ::= aSb | c
```

Matrices de las relaciones:

```
=. : 0 0 1 0    F  : 0 1 0 1   L : 0 0 1 1
     1 0 0 0         0 0 0 0      0 0 0 0
     0 0 0 0         0 0 0 0      0 0 0 0
     0 0 0 0         0 0 0 0      0 0 0 0
F+ = F
L+ = L
F* : 1 1 0 1
     0 1 0 0
     0 0 1 0
     0 0 0 1
```

Operando, sale:

```
<. : 0 0 0 0    >. : 0 0 0 0
     0 1 0 1         0 0 0 0
     0 0 0 0         0 0 1 0
     0 0 0 0         0 0 1 0
```

Con lo que la matriz de precedencia queda:

```
      S a b c -|
     ----------
   S |    =   >
   a |= <   < >
   b |    >   >
   c |    >   >
   |-|< < < <
```

Analicemos la instrucción: aacbb

```
Pila            Entrada
------------    ------------
|-             < aacbb-|
|-<a           < acbb-|
|-<a<a         < cbb-|
|-<a<a<c       > bb-|
|-<a<a=S       = bb-|
|-<a<a=S=b     > b-|
|-<a=S         = b-|
|-<a=S=b       > -|
|-<S             -|
```

Luego la cadena es aceptada.

Analicemos la instrucción: aabb

```
Pila            Entrada
------------    ------------
|-             < aabb-|
```

```
|-<a          < abb-|
|-<a<a          bb-|     (no hay relación)
```

Luego la cadena es rechazada.

Analicemos la instrucción: acbb

```
Pila            Entrada
------------    ------------
|-             < acbb-|
|-<a           < cbb-|
|-<a<c         > bb-|
|-<a=S         = bb-|
|-<a=S=b       > b-|
|-<S           = b-|
|-<S=b         > -|       (no hay regla)
```

Luego la cadena es rechazada.

Funciones de precedencia

La matriz de precedencias ocupa NxN. A veces se puede construir dos funciones de precedencia que ocupan sólo 2xN. (Linealización de la matriz). Estas funciones, de existir, no son únicas (hay infinitas).

Para el ejemplo anterior valen:

```
      |-  S  a  b  c  -|
   f  0  2  2  3  3
   g     2  3  2  3  0
```

Si se pueden construir f y g, se verifica que

- f(R)=g(S) => R =. S
- f(R)<g(S) => R <. S
- f(R)>g(S) => R >. S

Existen matrices no linearizables. Ejemplo:

```
   A B
 A  = >
 B  = =
```

pues debería ser

```
  f(A) = g(A)
  f(A) > g(B)
  f(B) = g(A)
  f(B) = g(B)
```

Es decir:

```
  f(A) > g(B) = f(B) = g(A) = f(A) => f(A) > f(A)
```

¡Contradicción!

Construcción de las funciones, cuando no hay inconsistencias:

- Se dibuja un grafo dirigido con 2N nodos llamados f1, f2, ... fN, g1, g2, ..., gN y un arco de fi a gj si Si >.= Sj y un arco de gj a fi si Si <.= Sj.
- A cada nodo se le asigna un número igual al número total de nodos accesibles desde él (incluido él mismo). El número asignado a fi es f(Si) y el asignado a gi es g(Si).

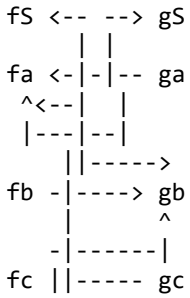
Prueba:

- Si Si =. Sj, hay una rama de fi a gj y viceversa, luego cualquier nodo accesible desde fi es accesible desde gj y viceversa. Luego f(Si)=g(Sj).
- Si Si >. Sj, hay una rama de fi a gj. Luego cualquier nodo accesible desde gj es accesible desde fi. Luego f(Si)>=g(Sj).

  Si f(Si)=g(Sj), existe un camino cerrado fi-gj-fk-gl-...-gm-fi, lo que implica que Si>.Sj, Sk<.=Sj, Sk>.=Sl, ... Si<.=Sm y reordenando: Si>.Sj>.=Sk>.=Sl>.=...>.=Sm>.=Si, es decir: f(Si)>f(Si), ¡contradicción!.

- Si Si <. Sj, la demostración es equivalente.

Ejemplo: grafo para obtener las funciones de precedencia del ejemplo anterior:



Mecanización del cálculo anterior: un grafo equivale a la matriz booleana de la relación xRy<=>hay un arco del nodo x al nodo y. Representamos el grafo por la matriz B 2Nx2N:

```
(0)     (>.=)
(<.=)'  (0)
```

Construimos B*. f(Si)=número de unos en la fila i. g(Si)=número de unos en la fila N+i.

En el ejemplo, B es la matriz

```
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
```

B* es la matriz

```
1 0 0 0 0 0 1 0
0 1 0 0 1 0 0 0
1 0 1 0 0 0 1 0
1 0 0 1 0 0 1 0
0 1 0 0 1 0 0 0
0 1 0 0 1 1 0 0
1 0 0 0 0 0 1 0
0 1 0 0 1 0 0 1
```

y las dos funciones f, g, salen como se dijo antes. Basta añadir los símbolos de principio y fin de cadena (con valor 0).

El uso de las funciones supone pérdida de información. Por ejemplo:

```
Pila           Entrada
------------   ------------
|-            < aabb-|
|-<a          < abb-|
|-<a<a        = bb-|
|-<a<a=b      > b-|      (no hay regla)
```

Ha habido que hacer un paso más.

Gramáticas de operadores

Tienen la siguiente restricción: en su conjunto de producciones no existe una regla de la forma:

```
U ::= xVWy
```

donde V,W en AN. En estas gramáticas, los símbolos terminales se llaman operadores, los no terminales operandos.

Relaciones de precedencia de operadores

Dada una gramática limpia de operadores, definimos las siguientes relaciones de precedencia entre los operadores: Para todo R,S en AT:

```
R =. S <=> existe U::=uRSv  en  P
         o existe U::=uRVSv  en  P, V en AN.
R <. S <=> existe U::=uRTv  en  P, T en AN,
           tal que T =>+ Sx
                 o T =>+ VSx, V en AN.
R >. S <=> existe U::=uTSv  en  P, T en AN,
           tal que T =>+ xR
                 o T =>+ xRV, V en AN.
R <.= S <=> R <. S ó R =. S
R >.= S <=> R >. S ó R =. S
```

Gramática de precedencia de operadores

Definición: *G es una G.P.O. si:*

1. *G es una gramática de operadores.*
2. *Sólo existe, como mucho, una relación de precedencia entre dos operadores cualesquiera.*

Construcción de las relaciones

1. La relación =. se construye por simple observación de las partes derechas de las reglas (definición de =.).
2. Definimos la relación FO ("First for Operators") así:

   U FO S <=> U::=Sx ó U::=VSx en P, U,V en AN, S en AT.

3. Definimos la relación LO ("Last for Operators") así:

   U LO S <=> U::=xS ó U::=xSV en P, U,V en AN, S en AT.

   Las dos relaciones anteriores se definen como matrices del mismo tamaño que las relaciones F y L, ya conocidas.

4. La relación <. se construye así:
5. `<. = (=.S) +.x (F*) +.x (FO)`

   donde +.x es el producto booleano de matrices y =.S es la relación =., tal como se definió para las gramáticas de precedencia simple.

6. La relación >. se construye así:
7. `>. = ((L*) +.x (LO))' +.x (=.S)`

   donde ' es la transposición de matrices.

Algoritmo de análisis

El equivalente del asidero se llama frase primaria. Es una frase que contiene al menos un símbolo terminal, pero no una frase menor que ella. Un teorema nos asegura que para toda frase primaria Si...Sj se cumple que

$$S_{i-1} <. S_i =. S_{i+1} =. \ldots =. S_{j-1} =. S_j >. S_{j+1}$$

Para reducir hacia el axioma tomamos la frase primaria situada más a la izquierda en la forma sentencial.

Ejemplo:

```
E ::= E+T | T
T ::= T*F | F
F ::= (E) | i | -F
```

La matriz de relaciones es:

	+	*	(	)	-
+	>	<	<	>	<
*	>	>	<	>	<
(	<	<	<	=	<
)	>	>		>	
-	>	>	<	>	<

El algoritmo es como el de las gramáticas de precedencia simple, pero se ignoran los símbolos no terminales.

Vamos a ver un algoritmo reducido que se emplea para el análisis y el cálculo de expresiones:

El algoritmo utiliza dos pilas: una de operadores, otra de operandos. La ventaja principal de estas gramáticas es que la matriz de las relaciones es mucho más pequeña.

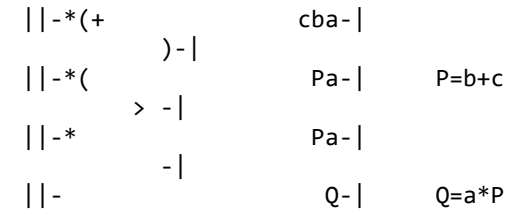
1. Se almacenan las producciones de la gramática en una tabla.
2. Se construye la matriz de precedencia de operadores, MP de dimensiones N.N (N=cardinal de AT), tal que

3. MP(i,j) = 0 si no existe relación entre Si y Sj
4. = 1 si Si <. Sj
5. = 2 si Si =. Sj
6. = 3 si Si >. Sj

7. Se inicializa la pila de operadores con el símbolo |- y se añade el símbolo -| al final de la cadena de entrada. La pila de operandos está inicialmente vacía.
8. Si el próximo símbolo de entrada es un identificador, se introduce en la pila de operandos. En caso contrario, se compara la precedencia del operador que esta en lo alto de la pila de operadores con el próximo operador.
9. Si no hay relación, error sintáctico: cadena rechazada.
10. Si los símbolos son (y), eliminar ambos.
11. Si hay la relación <. o la relación =., se introduce el símbolo de entrada en la pila de operadores y se avanza el puntero sobre la entrada. Volver al paso 4.
12. Si la relación es >., se llama la rutina semántica asociada al símbolo situado en lo alto de la pila, eliminándolo de ésta. Se eliminan también los operandos asociados y se introduce en la pila de operandos el resultado de la ejecución de la rutina semántica.
13. Si en la pila de operadores sólo queda |-, la cadena de entrada ha quedado reducida al símbolo -|, y la pila de operandos contiene un solo valor, la cadena ha quedado reconocida y el resultado de la expresión está en la pila de operandos.
14. Volver al paso 4.

Ejemplo: análisis de a*(b+c).

```
          a*(b+c)-|
||-                  |
       < *(b+c)-|
||-                a-|
       < (b+c)-|
||-*               a-|
          b+c)-|
||-*(              a-|
       < +c)-|
||-*(             ba-|
          c)-|
||-*(+            ba-|
       > )-|
```



Este algoritmo tiene problemas, como no localizar los errores en instrucciones como "a b +", "+b-c", etc. Se puede añadir una prueba para comprobar que no hay dos operandos seguidos, ni un operando seguido por función monádica, o se puede pasar a un algoritmo semejante al de las gramáticas de precedencia simple.

Gramáticas LR(k)

Informalmente, una gramática LR(k) (Knuth, 1965) permite analizar de forma determinista, de izquierda a derecha, una cadena de entrada, observando a lo sumo los siguientes elementos:

- El contexto izquierdo (ya analizado).
- La parte de la cadena que se va a reducir.
- Un máximo de k símbolos del contexto derecho.

Los analizadores basados en gramáticas LR(k) presentan características comunes con los analizadores "top-down" y "bottom-up".

Configuración

Es una producción con un marcador, que indica hasta donde hemos llegado en el análisis de la parte derecha. Por ejemplo, sea la gramática parcial

```
1:  <Bloque> ::= begin <Decs> ; <Ejecs> end
2:  <Decs>   ::= <Dec>
3:             |  <Decs> ; <Dec>
4:  <Ejecs>  ::= <Ejec>
5:             |  <Ejec> ; <Ejecs>
```

Son configuraciones válidas las siguientes:

```
(1,0)
(2,1)
(3,3)
```

donde (a,b) indica que estamos analizando la posición b (en origen 0) de la parte derecha de la regla de producción número a.

Estado

Se representa mediante un conjunto de configuraciones. Indica las configuraciones posibles en la situación actual del análisis. Su número es finito.

Supongamos que estamos en la configuración representada por (a,b), y que el símbolo siguiente a la posición b en la parte derecha de la regla a es el símbolo no terminal U. Sean c,...,d las reglas cuya parte izquierda es U. Entonces, de la configuración (a,b) se podrá pasar a las configuraciones (c,0),...,(d,0). Si aplicamos el mismo procedimiento indefinidamente, hasta que no aparezcan configuraciones nuevas, tendremos la "clausura" de la configuración (a,b): el conjunto de estados que han ido apareciendo, incluido el (a,b). Si partimos de un conjunto de configuraciones (un estado), la clausura correspondiente será igual a la unión de las clausuras de las configuraciones del estado.

Estado inicial

Para preparar el análisis LR(k) de una gramática hacemos inicialmente dos cosas:

- Añadimos a la gramática la regla nueva:
- `S' ::= S -|`

  donde S' es un símbolo no terminal nuevo (nuevo axioma), S es el axioma de la gramática original, y -| es un símbolo terminal nuevo (símbolo final de cadena) que, además, añadimos al final de la cadena de entrada. Esta regla pasa a ser la regla número 0 de la nueva gramática. Las demás reglas quedan como estaban.

- El estado inicial del análisis es la clausura de {(0,0)}.

Operación corrimiento

Dado un estado, se puede pasar al estado sucesor mediante la operación "corrimiento", que consiste en avanzar un símbolo sobre la cadena de entrada, pasando además del estado actual a otro nuevo, que se obtiene así:

- Se extraen del estado las configuraciones (a,b) en las que el símbolo siguiente a la posición b en la parte derecha de la regla a es el próximo símbolo de la cadena de entrada. El estado sucesor es la clausura del conjunto de configuraciones {(a,b+1)}.

Obsérvese que en la cadena de entrada podemos encontrar símbolos terminales o no terminales, como veremos a continuación.

Operación reducción

Cuando una configuración representa el final de la parte derecha de una regla, se puede aplicar esa regla, añadiendo su parte izquierda a la izquierda de la cadena de entrada, eliminando de la lista de estados por los que se ha pasado los que corresponden a la parte derecha de la regla aplicada, y volviendo al estado anterior al comienzo del análisis de dicha regla.

Tabla del análisis

La tabla del análisis tiene tantas filas como estados accesibles a partir del inicial, y tantas columnas como símbolos del vocabulario.

Para construir la tabla del análisis, partimos del estado inicial y calculamos los estados accesibles a partir de él, los estados accesibles a partir de éstos, etc., hasta que no salgan estado nuevos, utilizando únicamente la operación corrimiento. A continuación, rellenamos algunas casillas con la operación reducción, utilizando para ello una regla que veremos después.

Además, pondremos "Fin" en la casilla cuya fila es el estado inicial y cuya columna es el axioma. Esto sólo podremos hacerlo en algunas gramáticas, en aquéllas en las que el axioma no puede aparecer en ninguna forma sentencial. Más adelante veremos cómo se actúa cuando esto no es cierto.

Ejemplo

Sea el ejemplo de la gramática que hemos visto antes. Consideraremos terminales a los símbolos <Dec> y <Ejec>.

Nuestra tabla inicial es:

```
       Bloque Decs Ejecs Dec Ejec begin  ;  end  -|
 S0      Fin
```

donde S0=clausura {(0,0)} = {(0,0),(1,0)}.

En el estado inicial, ignoraremos la configuración (0,0), cuya casilla ya está rellena. La otra, (1,0), es anterior a un símbolo terminal: "begin". Luego el único estado accesible desde el inicial es la clausura de {(1,1)} = {(1,1),(2,0),(3,0)} = S1. La tabla queda:

```
       Bloque Decs Ejecs Dec Ejec begin  ;  end  -|
 S0      Fin                      S1
 S1
```

A partir de S1 podemos encontrar en la cadena de entrada los símbolos <Dec> y <Decs>. <Dec> nos pasa a la configuración (2,1), <Decs> a la {(1,2),(3,1)}. Tendremos, pues, los estados S2=clausura {(2,1)}={(2,1)} y S3=clausura {(1,2),(3,1)}={(1,2),(3,1)}. La tabla queda:

```
       Bloque Decs Ejecs Dec Ejec begin  ;  end  -|
 S0      Fin                      S1
 S1           S3        S2
 S2
 S3
```

Desde S2 ya no se puede seguir: es fin de regla. Desde S3 sí. En ambos casos, el símbolo siguiente sólo puede ser ";". Las configuraciones siguientes son: {(1,3),(3,2)}. El estado siguiente será su clausura: S4={(1,3),(3,2),(4,0),(5,0)}. La tabla queda:

	Bloque	Decs	Ejecs	Dec	Ejec	begin	;	end	-|
S0	Fin				S1				
S1		S3		S2					
S2									
S3						S4			
S4									

Partiendo de S4, podemos encontrar los símbolos <Ejecs>, <Dec> y <Ejec>. Tendremos, por tanto, tres transiciones posibles. La primera nos lleva a {(1,4)}, cuya clausura es él mismo. Lo llamaremos S5. La segunda nos lleva a {(3,3)}, cuya clausura es él mismo. Lo llamaremos S6. La tercera nos lleva a {(4,1),(5,1)}, cuya clausura es él mismo. Lo llamaremos S7. La tabla queda:

	Bloque	Decs	Ejecs	Dec	Ejec	begin	;	end	-|
S0	Fin				S1				
S1		S3		S2					
S2									
S3						S4			
S4			S5	S6	S7				
S5									
S6									
S7									

Desde S5 sólo podemos encontrar el símbolo "end", que nos lleva a {(1,5)}, cuya clausura es él mismo. Lo llamamos S8. Desde S6 no podemos llegar a ningún sitio: es fin de regla. Desde S7 podemos encontar el símbolo ";" y llegar a {(5,2)}, cuya clausura es {(5,2),(4,0),(5,0)}, un estado nuevo. Lo llamamos S9. La tabla queda:

	Bloque	Decs	Ejecs	Dec	Ejec	begin	;	end	-|
S0	Fin				S1				
S1		S3		S2					
S2									
S3						S4			
S4			S5	S6	S7				
S5							S8		
S6									
S7						S9			
S8									
S9									

Desde S8 no podemos seguir: es fin de regla. Desde S9 podemos encontrar <Ejecs> o <Ejec>. El primero nos lleva a {(5,3)}, cuya clausura es él mismo. Lo llamamos S10. El segundo nos lleva a {(4,1),(5,1)}, es decir, a S7. La tabla queda:

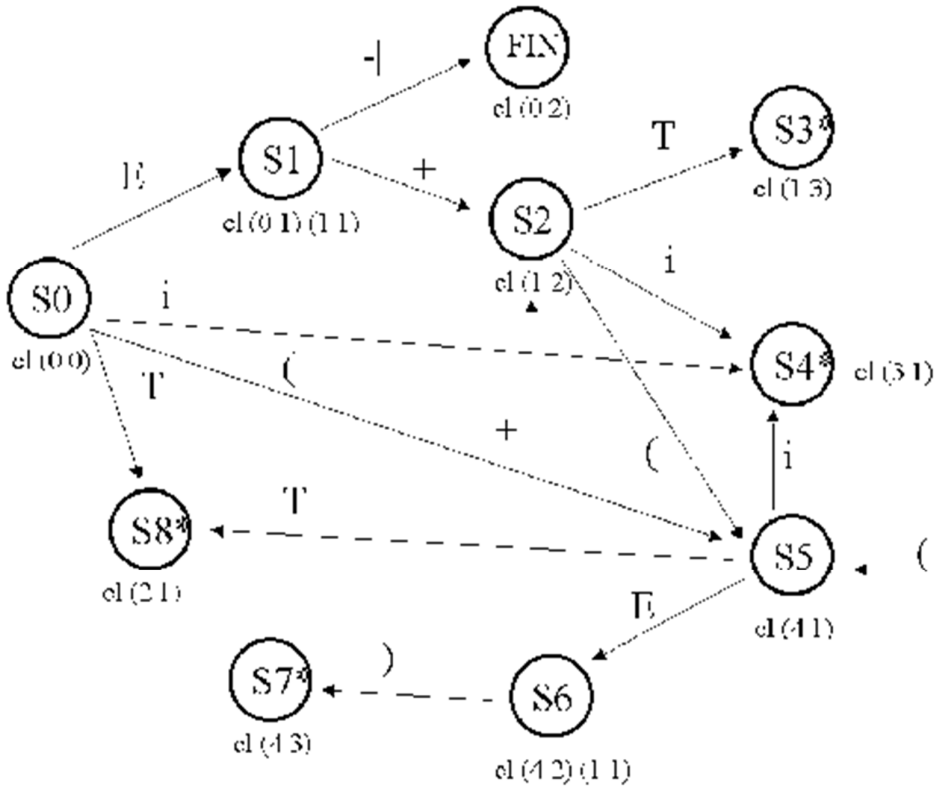
	Bloque	Decs	Ejecs	Dec	Ejec	begin	;	end	-|
S0	Fin				S1				
S1		S3		S2					
S2									
S3						S4			
S4			S5	S6	S7				
S5							S8		
S6									
S7						S9			
S8									
S9			S10		S7				
S10									

Desde S10 no podemos seguir: es fin de regla. Por tanto, ya no hay más estados. La tabla está completa, salvo por la operación reducción.

Resumiendo: los estados encontrados son:

```
   S0   {(0,0),(1,0)}
   S1   {(1,1),(2,0),(3,0)}
  *S2   {(2,1)}
   S3   {(1,2),(3,1)}
   S4   {(1,3),(3,2),(4,0),(5,0)}
   S5   {(1,4)}
  *S6   {(3,3)}
  *S7   {(4,1),(5,1)}
  *S8   {(1,5)}
   S9   {(5,2),(4,0),(5,0)}
  *S10  {(5,3)}
```

donde los estados que contienen una configuración final de regla aparecen señalados con un asterisco.



Reglas para la operación reducción

Dependiendo de la regla utilizada para rellenar algunas casillas de la tabla con la operación reducción, tendremos distintos tipos de familias de analizadores LR(k).

Analizadores LR(0)

La regla es: "si un estado contiene configuraciones de final de regla, reduciremos esa regla en todas las casillas correspondentes a un símbolo terminal".

La aplicación de la regla anterior puede provocar conflictos de dos tipos:

- Si un estado contiene una configuración final de regla y otra que no lo es, situada delante de un símbolo terminal, deberíamos poner dos transiciones en la misma casilla, lo que la haría no determinista. Este es un conflicto corrimiento-reducción.
- Si un estado contiene dos configuraciones final de regla, también deberíamos poner dos transiciones en la misma casilla. Este es un conflicto reducción-reducción.

La aplicación de esta regla al ejemplo anterior nos daría la siguiente tabla:

	Bloque	Decs	Ejecs	Dec	Ejec	begin	;	end	-|
S0		Fin				S1			
S1			S3		S2				
S2					R2	R2	R2	R2	R2
S3							S4		
S4				S5	S6	S7			
S5								S8	
S6					R3	R3	R3	R3	R3
S7					R4	R4	R4	R4/S9	R4
S8					R1	R1	R1	R1	R1
S9				S10		S7			
S10					R5	R5	R5	R5	R5

Tenemos, pues un conflicto corrimiento-reducción. Por tanto, esta gramática no es LR(0).

Si al aplicar la regla anterior no hay ningún conflicto, decimos que la gramática está en forma LR(0). Por ejemplo, sea la gramática:

```
1:  E ::= E + T
2:  E ::= T
3:  T ::= i
4:  T ::= ( E )
```

Añadimos la regla 0:

```
0:  S ::= E -|
```

Aplicando la técnica anterior, obtenemos la siguiente tabla de análisis:

	E	T	i	+	(	)	-|
S0	S1	S8	S4		S5		
S1				S2			Fin
S2		S3	S4		S5		
S3			R1	R1	R1	R1	R1
S4			R3	R3	R3	R3	R3
S5	S6	S8	S4		S5		
S6				S2		S7	
S7			R4	R4	R4	R4	R4
S8			R2	R2	R2	R2	R2

Obsérvese que, en este caso, el axioma sí puede aparecer como parte de una forma sentencial, por lo que hemos tenido que introducir un estado

más (S1), tratando la configuración (0,0) como otra cualquiera. La situación de aceptación, señalada por Fin, tiene lugar en la configuración (0,1), que aparece en dicho estado nuevo cuando se localiza el símbolo de fin de cadena.

Esta gramática deja de ser LR(0) si se introduce la precedencia de operadores. También surgen conflictos si existen reglas que generan la cadena vacía, especialmente cuando hay otras reglas con la misma parte izquierda.

Analizadores SLR(1)

La regla es: "si un estado contiene configuraciones de final de regla, reduciremos esa regla en todas las casillas correspondentes a un símbolo terminal que aparezca detrás de la parte izquierda de esa regla (directamente o después de alguna derivación) en la parte derecha de otra".

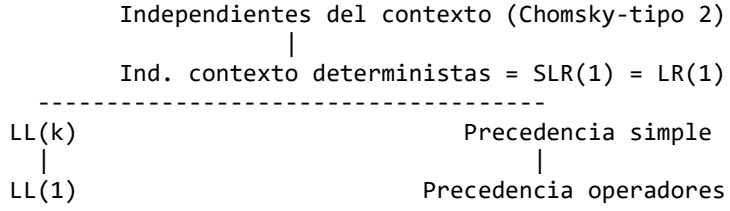
La aplicación de esta regla al ejemplo anterior nos daría la siguiente tabla:

	Bloque	Decs	Ejecs	Dec	Ejec	begin	;	end	-	|
S0	Fin				S1					
S1		S3		S2						
S2							R2			
S3							S4			
S4			S5	S6	S7					
S5								S8		
S6							R3			
S7							S9	R4		
S8									R1	
S9			S10		S7					
S10								R5		

Si al aplicar la regla anterior no hay ningún conflicto, decimos que la gramática está en forma SLR(1). Para cualquier lenguaje independiente del contexto determinista existe una gramática SLR(1) que lo representa.

En las gramáticas LR(1), cada configuración se incrementa con información sobre los símbolos de entrada que pueden venir a continuación. Esto complica los estados y las transiciones, pero en general no es necesario. Se demuestra que todo lenguaje representable mediante una gramática LR(1) también puede representarse mediante una gramática SLR(1).

Relaciones entre lenguajes



Análisis

Para realizar el análisis, aplicaremos el siguiente algoritmo:

1. Se construye la tabla del análisis, como se ha indicado.
2. Se añade a la cadena de entrada el símbolo final.
3. Se construye una pila de estados, inicializada al estado inicial.
4. En cada paso, el estado actual en que se encuentra el análisis es el que está en lo alto de la pila. Se indexa la tabla del análisis, por filas por el estado actual, por columnas por el primer símbolo de la cadena de entrada, para hallar la acción a realizar.
5. Si es un corrimiento, eliminamos el primer símbolo de la cadena de entrada, introducimos el nuevo estado en lo alto de la pila y volvemos al paso 4.
6. Si es una reducción, eliminamos de la pila tantos estados como símbolos haya en la parte derecha de la regla a aplicar, añadimos la parte izquierda de la regla a la izquierda de la cadena de entrada y volvemos al paso 4.
7. Si es Fin, la cadena de entrada estará constituida por el axioma de la gramática, seguido por el símbolo final de cadena. La cadena inicial ha sido aceptada (es correcta).
8. Si está en blanco, la cadena inicial de entrada es errónea.

Utilizaremos la tabla y la gramática de nuestro ejemplo para analizar la cadena de entrada:

```
begin <Dec> ; <Dec> ; <Ejec> ; <Ejec> end
```

Partimos del estado inicial S0 = {(0,0),(1,0)}. La sucesión de estados por los que pasamos es:

```
Estados  Cadena de entrada                          Acción
-------  ------------------------------------------ ------
S0       begin <Dec> ; <Dec> ; <Ejec> ; <Ejec> end -| S1

S1       <Dec> ; <Dec> ; <Ejec> ; <Ejec> end -|       S2
S0

S2       ; <Dec> ; <Ejec> ; <Ejec> end -|             R2
S1
S0

S1       <Decs> ; <Dec> ; <Ejec> ; <Ejec> end -|      S3
S0

S3       ; <Dec> ; <Ejec> ; <Ejec> end -|             S4
S1
S0

S4       <Dec> ; <Ejec> ; <Ejec> end -|               S6
S3
S1
S0

S6       ; <Ejec> ; <Ejec> end -|                     R3
S4
```

```
S3
S1
S0

S1      <Decs> ; <Ejec> ; <Ejec> end -|          S3
S0

S3      ; <Ejec> ; <Ejec> end -|                 S4
S1
S0

S4      <Ejec> ; <Ejec> end -|                   S7
S3
S1
S0

S7      ; <Ejec> end -|                          S9
S4
S3
S1
S0

S9      <Ejec> end -|                            S7
S7
S4
S3
S1
S0

S7      end -|                                   R4
S9
S7
S4
S3
S1
S0

S9      <Ejecs> end -|                           S10
S7
S4
S3
S1
S0

S10     end -|                                   R5
S9
S7
S4
S3
S1
S0

S4      <Ejecs> end -|                           S5
S3
S1
S0

S5      end -|                                   S8
S4
S3
S1
S0

S8      -|                                       R1
S5
S4
S3
```

```
  S1
  S0

  S0       <Bloque> -|                                  Fin
```

Capítulo 5. Análisis semántico

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológico y sintáctico.

El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

En compiladores de un solo paso, las llamadas a las rutinas semánticas se realizan directamente desde el analizador sintáctico y son dichas rutinas las que llaman al generador de código. El instrumento más utilizado para conseguirlo es la gramática de atributos.

En compiladores de dos o más pasos, el análisis semántico se realiza independientemente de la generación de código, pasándose información a través de un archivo intermedio, que normalmente contiene información sobre el árbol sintáctico en forma linealizada (para facilitar su manejo y hacer posible su almacenamiento en memoria auxiliar).

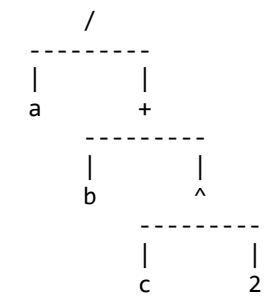
En cualquier caso, las rutinas semánticas suelen hacer uso de una pila (la pila semántica) que contiene la información semántica asociada a los operandos (y a veces a los operadores) en forma de *registros semánticos*.

Propagación de atributos

Sea la expresión

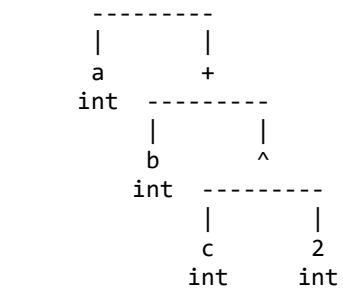
```
int a,b,c;
a/(b+c^2)
```

El árbol sintáctico es:

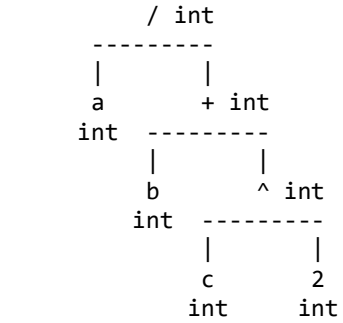


De la instrucción declarativa, la tabla de símbolos y el analizador morfológico obtenemos los atributos de los operandos:

```
        /
```



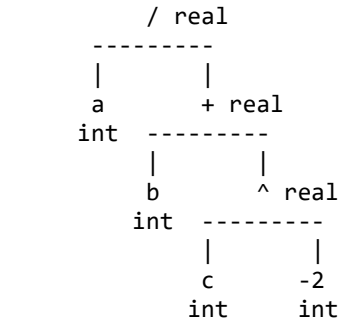
Propagando los atributos obtenemos:



Si la expresión hubiera sido

```
a/(b+c^-2)
```

El árbol sintáctico sería el mismo, sustituyendo 2 por -2. Sin embargo, la propagación de atributos sería diferente:

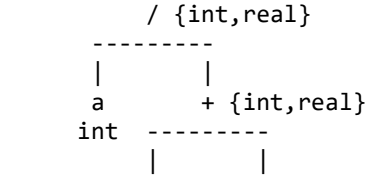


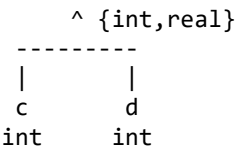
En algún caso podría llegar a producirse error (p.e. si / representara sólo la división entera).

Si la expresión hubiera sido

```
int a,b,c,d;
a/(b+c^d)
```

El árbol sintáctico sería el mismo, sustituyendo 2 por d. Sin embargo, la propagación de atributos sería incompleta:





El analizador semántico podría reducir los tipos inseguros al tipo máximo (real) o utilizar un tipo interno nuevo (ej. arit={int,real}, una unión).

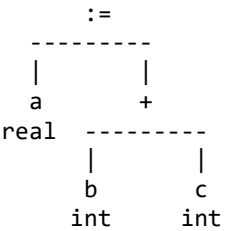
Lo anterior es un ejemplo de propagación bottom-up. La propagación top-down también es posible: lo que se transmite son las restricciones y los tipos de las hojas sirven de comprobación. Por ejemplo, si la división sólo puede ser entera, transmitimos hacia abajo la restricción de que sus operandos sólo pueden ser enteros. Al llegar a d, esa restricción se convierte en que d debe ser positiva. Si no lo es, error.

La implantación de todos los casos posibles de operación con tipos mixtos podría ser excesivamente cara. En su lugar, se parte de operaciones relativamente simples (ej. int+int, real+real) y no se implementan las restantes (ej. int+real, real+int), añadiendo en su lugar operaciones monádicas de cambio de tipo (ej. int->real).

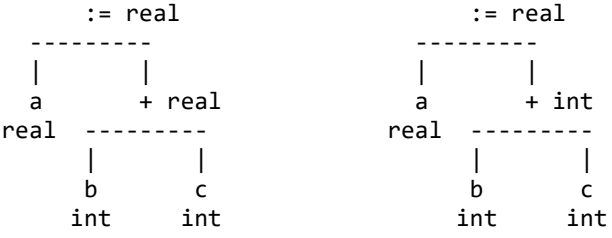
Esta decisión puede introducir ambigüedades. Por ejemplo, sea el programa

```
real a;
int b,c;
a:=b+c
```

El árbol sintáctico es:



Existen dos conversiones posibles:



El problema es que no tenemos garantía de que los dos procedimientos sean equivalentes. El segundo puede dar overflow, el primero pérdida de precisión. La definición del lenguaje debe especificar estos casos.

Las transformaciones posibles se pueden representar mediante un grafo cuyos nodos son los tipos de datos y cada arco indica una transformación. Dado un operando de tipo A que se desea convertir al tipo B, se trata de encontrar una cadena de arcos que pase de A a B en el grafo anterior. Podría haber varios grafos, cada uno de los cuales se aplicará en diferentes condiciones, por ejemplo, uno para las asignaciones, otro para las expresiones, etc.

Gramática de atributos

Es una extensión de la notación de Backus que consiste en introducir en las reglas sintácticas ciertos símbolos adicionales no sintácticos (*símbolos de acción*) que, en definitiva, se reducen a llamadas implícitas o explícitas a rutinas semánticas.

Por ejemplo: sea la gramática simplificada que analiza las instrucciones de asignación:

```
<AsSt> ::= id #PId := <Exp> #RAs
<Exp>  ::= <Exp> + <T> #RS | <T>
<T>    ::= id #PId | Ct #PCt
```

Se observará que hemos hecho uso de cuatro símbolos de acción:

- **#PId**: PUSH a la pila semántica el registro asociado al identificador.
- **#PCt**: PUSH a la pila semántica el registro asociado a la constante.
- **#RS**: Realizar suma: POP los dos registros superiores de la pila; comprobar que es posible sumarlos; realizar la suma (mediante una llamada al generador de código) o generar representación intermedia (si es una compilación en dos o más pasos); PUSH registro semántico del resultado en la pila semántica.
- **#RAs**: Realizar asignación: POP los dos registros superiores de la pila; comprobar que es posible realizar la asignación; realizarla (mediante una llamada al generador de código) o generar representación intermedia (si es una compilación en dos o más pasos).

En los analizadores sintácticos top-down basados en gramáticas LL(1), la introducción de los símbolos de acción en las rutinas correspondientes es trivial. En los analizadores bottom-up basados en gramáticas SLR(1) es más delicado, pues los estados del análisis se mueven simultáneamente sobre varias reglas. Sólo en el momento de realizar una reducción sabemos exactamente dónde estamos. Por ello, se suele introducir la restricción de que los símbolos de acción deben estar situados únicamente al final de una regla. Cuando no se cumple esto (como en el ejemplo anterior) es trivial conseguirlo, introduciendo símbolos no terminales adicionales. Algunos generadores de analizadores sintácticos (como YACC) lo realizan automáticamente. En nuestro ejemplo, quedaría:

```
<AsSt> ::= <ASPI> := <Exp> #RAs
```

```
<ASPI> ::= id #PId
<Exp>  ::= <Exp> + <T> #RS | <T>
<T>    ::= id #PId | Ct #PCt
```

Poner un ejemplo del análisis sintáctico-semántico bottom-up de la instrucción A := B + 1.

Otro ejemplo: instrucciones condicionales con las reglas

```
<Instr> ::= If <Expr> #S2 then <Instr> #S1 |
            If <Expr> #S2 then <Instr> else #S3 <Instr> #S1
```

Más adelante se verá cuáles son las tres acciones semánticas. Para que todas queden al final de una regla, basta cambiar estas reglas por:

```
<Instr> ::= If <Expr1> then <Instr> #S1 |
            If <Expr1> then <Instr> <Else> <Instr> #S1
<Expr1> ::= <Expr> #S2
<Else>  ::= else #S3
```

Generación de representaciones intermedias

Existen dos representaciones intermedias principales:

- Notación sufija
- Cuádruplas

Los operadores diádicos (o binarios) pueden especificarse mediante tres notaciones principales:

- Prefija: el operador diádico es analizado antes que sus operandos.
- Infija: el operador diádico es analizado entre sus operandos.
- Sufija: el operador diádico es analizado después que sus operandos.

En los lenguajes de programación clásicos, los operadores diádicos se representan usualmente en notación infija. La notación prefija permite al operador influir sobre la manera en que se procesan sus operandos, pero a cambio suele exigir mucha más memoria. La sufija no permite esa influencia, pero es óptima en proceso de memoria y permite eliminar el procesado de los paréntesis.

Los operadores monádicos sólo pueden presentarse en notación prefija o sufija.

Además, un árbol sintáctico puede representarse en forma de tuplas de n elementos, de la forma (operador, operando-1, ..., operando-k, nombre). Las tuplas pueden tener longitud variable o fija (con operandos nulos). Las más típicas son las cuádruplas, aunque éstas pueden representarse también en forma de tripletes.

Notación sufija

Llamada también postfija o polaca inversa, se usa para representar expresiones sin necesidad de paréntesis.

Ejemplos:

```
a*b          ab*
a*(b+c/d)    abcd/+*
a*b+c*d      ab*cd*+
```

Los identificadores aparecen en el mismo orden. Los operadores en el de evaluación (de izquierda a derecha).

Problema: operadores monádicos (unarios). O bien se transforman en diádicos (binarios) o se cambia el símbolo.

Ejemplo: -a se convierte en 0-a o en @a

```
a*(-b+c/d)   ab@cd/+*
```

Existen dos problemas principales:

- Construir la notación sufija a partir de la infija.
- Analizar la notación sufija en el segundo paso de la compilación.

Rutina semántica para transformar de infijo a sufijo

Si el analizador sintáctico es bottom-up, hacemos la siguiente suposición: "Cuando aparece un no terminal V en el asidero, la cadena polaca correspondiente a la subcadena que se redujo a V ya ha sido generada".

Se utiliza una pila donde se genera la salida, inicialmente vacía. Las acciones semánticas asociadas a las reglas son:

```
E ::= E + T          Push +
E ::= E - T          Push -
E ::= T
T ::= T * F          Push *
T ::= T / F          Push /
T ::= F
F ::= i              Push i
F ::= (E)
F ::= - F            Push @
```

Análisis de la notación sufija

La gramática completa que permite analizar la notación sufija es:

```
<Operando> ::= id |
               cte |
               <Operando> <Operando> <Operador diádico> |
               <Operando> <Operador monádico>
<Operador diádico> ::= + | - | * | / | ...
<Operador monádico> ::= @ | ...
```

Algoritmo de evaluación de una expresión en notación sufija que utiliza una pila:

- Si el próximo símbolo es un identificador, se pasa a la pila. Corresponde a la aplicación de la regla
- `<Operando> ::= id`
- Si el próximo símbolo es una constante, se pasa a la pila. Corresponde a la aplicación de la regla
- `<Operando> ::= cte`
- Si el próximo símbolo es un operador diádico, se aplica el operador a los dos operandos situados en lo alto de la pila y se sustituyen éstos por el resultado de la operación. Corresponde a la aplicación de la regla
- `<Operando> ::= <Operando> <Operando> <Operador diádico>`
- Si el próximo símbolo es un operador monádico, se aplica el operador al operando situado en lo alto de la pila y se sustituye éste por el resultado de la operación. Corresponde a la aplicación de la regla
- `<Operando> ::= <Operando> <Operador monádico>`

Ejemplo: calcular ab@cd/+*.

Extensión de la notación sufija a otros operadores

- La asignación, teniendo en cuenta que podemos no querer valor resultante. Además, no interesa tener en la pila el valor del identificador izquierdo, sino su dirección.
- `a:=b*c+d` `abc*d+:=`
- La transferencia (GOTO).
- `GOTO L` `L TR`
- La instrucción condicional
- `if p then inst1 else inst2`

  se convierte en

  ```
  p L1 TRZ inst1 L2 TR inst2
                     L1:   L2:
  ```

- Subíndices:
- `a[exp1; exp2; ...; expn]`

  se convierte en

  ```
  a exp1 exp2 ... expn SUBIN-n
  ```

Cuádruplas

Una operación diádica se puede representar mediante la cuádrupla

```
(<Operador>, <Operando1>, <Operando2>, <Resultado>)
```

Ejemplo:

```
(*,A,B,T)
```

Una expresión se puede representar mediante un conjunto de cuádruplas. Ejemplo: la expresión a*b+c*d equivale a:

```
(*,a,b,t1)
(*,c,d,t2)
(+,t1,t2,t3)
```

Ejemplo: la expresión c:=a[i;b[j]] equivale a:

```
(*,i,d1,t1)
(+,t1,b[j],t2)
(:=,a[t2],,c)
```

Tripletes

No se pone el resultado, se sustituye por referencias a tripletes. Por ejemplo: la expresión a*b+c*d equivale a:

```
(1) (*,a,b)
(2) (*,c,d)
(3) (+,(1),(2))
```

mientras que a*b+1 equivale a:

```
(1) (*,a,b)
(2) (*,(1),1)
```

Tripletes indirectos: se numeran arbitrariamente los tripletes y se da el orden de ejecución. Ejemplo, sean las instrucciones:

```
a := b*c
b := b*c
```

Equivalen a los tripletes

```
(1) (*,b,c)
(2) (:=,(1),a)
(3) (:=,(1),b)
```

y el orden de ejecución es (1),(2),(1),(3). Esta forma es útil para preparar la optimización de código. Si hay que alterar el orden de las operaciones o eliminar alguna, es más fácil hacerlo ahí.

Generación automática de cuádruplas

En un análisis bottom-up, asociamos a cada símbolo no terminal una información semántica, y a cada regla de producción una acción semántica. Ejemplo, sea la gramática

```
E ::= E + T
E ::= E - T
E ::= T
T ::= T * F
T ::= T / F
T ::= F
F ::= i
F ::= (E)
F ::= -F
```

La regla F::=i asocia a F como información semántica el identificador concreto.

La regla F::=(E) asocia a F como información semántica la información semántica asociada a E.

La regla U::=V asocia a U como información semántica la información semántica asociada a V.

La regla U::=VoW analiza la compatibilidad de los operandos, crea la cuádrupla (o,Sem(V),Sem(W),Ti) y asocia a U la información semántica Ti.

La regla U::=oV crea la cuádrupla (o,Sem(V),,Ti) y asocia a U la información semántica Ti.

La información semántica se suele almacenar en otra pila paralela.

Ejemplo: análisis de a*(b+c)

```
Pila            Entrada        Regla        Cuádrupla
----------      -------------  ------------ ---------
|               a*(b+c)|
|a              *(b+c)|        F::=i
|F(a)           *(b+c)|        T::=F
|T(a)           *(b+c)|
|T(a)*          (b+c)|
|T(a)*(         b+c)|
|T(a)*(b        +c)|           F::=i
|T(a)*(F(b)     +c)|           T::=F
|T(a)*(T(b)     +c)|           E::=T
|T(a)*(E(b)     +c)|
|T(a)*(E(b)+    c)|
|T(a)*(E(b)+c    )|             F::=i
|T(a)*(E(b)+F(c) )|             T::=F
|T(a)*(E(b)+T(c) )|            E::=E+T
|T(a)*(E(b)+T(c) )|            E::=E+T      (+,b,c,T1)
|T(a)*(E(T1)     )|
|T(a)*(E(T1))    |             F::=(E)
|T(a)*F(T1)      |             T::=T*F      (*,a,T1,T2)
|T(T2)           |             E::=T
|E(T2)           |
```

Ejemplo de generación de cuádruplas en análisis top-down

```
unsigned int E (char *cadena, unsigned int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case 'i':
      push (id);
      i++;
      i = V (cadena, i);
      break;
    case '(':
      i++;
      i = E (cadena, i);
      i = C (cadena, i);
      i = V (cadena, i);
      break;
    default: return -1;
```

```
      }
  return i;
}

unsigned int V (char *cadena, unsigned int i)
{
  unsigned int j;
  if (i<0) return i;
  switch (cadena[i]) {
    case '*':
    case '/':
      j = i;
      i++;
      i = T (cadena, i);
      cuad (cadena[j], pop(), pop(), gen(Ti));
      push (Ti);
      i = X (cadena, i);
      break;
    case '+':
    case '-':
      j = i;
      i++;
      i = E (cadena, i);
      cuad (cadena[j], pop(), pop(), gen(Ti));
      push (Ti);
      break;
      }
  return i;
}

unsigned int X (char *cadena, unsigned int i)
{
  unsigned int j;
  if (i<0) return i;
  switch (cadena[i]) {
    case '+':
    case '-':
      j = i;
      i++;
      i = E (cadena, i);
      cuad (cadena[j], pop(), pop(), gen(Ti));
      push (Ti);
      break;
      }
  return i;
}

unsigned int T (char *cadena, unsigned int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case 'i':
      push (id);
      i++;
      i = U (cadena, i);
      break;
    case '(':
      i++;
      i = E (cadena, i);
      i = C (cadena, i);
      i = U (cadena, i);
      break;
    default: return -2;
      }
  return i;
}
```

```
unsigned int U (char *cadena, unsigned int i)
{
  if (i<0) return i;
  unsigned int j;
  switch (cadena[i]) {
    case '*':
    case '/':
      j = i;
      i++;
      i = T (cadena, i);
      cuad (cadena[j], pop(), pop(), gen(Ti));
      push (Ti);
      break;
      }
  return i;
}

unsigned int F (char *cadena, unsigned int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case 'i':
      push (id);
      i++;
      break;
    case '(':
      i++;
      i = E (cadena, i);
      i = C (cadena, i);
      break;
    default: return -3;
      }
  return i;
}

unsigned int C (char *cadena, unsigned int i)
{
  if (i<0) return i;
  switch (cadena[i]) {
    case ')':
      i++;
      break;
    default: return -4;
      }
  return i;
}
```

Semántica de instrucciones condicionales

Sean las reglas

```
   <Instr> ::= If <Expr> S2 then <Instr> S1 |
               If <Expr> S2 then <Instr> else S3 <Instr> S1
```

Una secuencia de cuádruplas equivalente a "If E1 then I1 else I2"

```
(p-1)  (?,?,?,t1)         Análisis de E1
(p)    (TRZ,(q+1),t1,)    S2: Crear cuádrupla (p) | Push p
       ...                Análisis de I1
(q)    (TR,(r),,)         S3: Crear cuádrupla (q)
                              Poner (cl.sig.) en top | Pop
                              Push (q)
(q+1)  ...                Análisis de I2
(r)                       S1: Poner (cl.sig.) en top | Pop
```

Una secuencia de cuádruplas equivalente a "If E1 then I1"

```
(p-1)  (?,?,?,t1)        Análisis de E1
(p)    (TRZ,(r),t1,)     S2: Crear cuádrupla (p) | Push p
       ...               Análisis de I1
(r)                      S1: Poner (cl.sig.) en top | Pop
```

Al generar la cuádrupla (p) no conocemos el valor de (q). Guardamos en una pila el número de la cuádrupla asociada y lo rellenamos más tarde, como indican los ejemplos.

Semántica de etiquetas y GOTO

Suponemos que las etiquetas aparecen en la tabla de símbolos con tres valores asociados: (tipo=etiqueta, bit=declarada/no declarada, número de cuádrupla).

Sea la regla

```
<Instr> ::= id : <Instr>
```

Semántica asociada:

```
{
Buscar id en la tabla de símbolos;
if (no está)
  Insertar id,valor=(etiqueta, declarada, cuádrupla siguiente);
else {
  if (tipo==etiqueta && bit==no declarada) {
    i=número de cuádrupla;
    while (i) {
      j=cuádrupla[i][2];
      cuádrupla[i][2]=cuádrupla siguiente;
      i=j;
      }
    Cambiar valor a (etiqueta, declarada, cuádrupla siguiente);
    }
  else error();
  }
}
```

Sea la regla

```
<Instr> ::= GOTO id
```

Semántica asociada:

```
{
Buscar id en la tabla de símbolos;
if (no está) {
  Insertar id,valor=(etiqueta, no declarada, cuádr.siguiente);
  Generar cuádrupla (TR,,,);
  }
else {
  if (tipo==etiqueta) {
    if (bit==declarada)
      Generar cuádrupla (TR,número de cuádrupla,,);
    else if (bit==no declarada) {
      i=número de cuádrupla;
      Cambiar valor a (etiqueta, no declarada, cuádr.siguiente);
```

```
        Generar cuádrupla (TR,i,,);
        }
      }
    else error();
    }
  }
```

Si se permiten etiquetas locales a bloques, podemos encontrar el siguiente caso:

```
L:  ...
    {
      ...
      GOTO L;
      ...
```

Tenemos ambigüedad: GOTO L puede ir a la etiqueta externa (ya definida o no) o a una etiqueta local al bloque posterior a la instrucción. Tenemos tres posibilidades:

- Un compilador en dos pasos.
- Forzar declaraciones de etiquetas.
- Tratar L en el bloque como si fuera local. Si al final del bloque descubrimos que no ha sido declarada, tratarla como si fuera global. La lista de referencias debería fundirse con la de L global (si L global no ha sido definida aún) o rellenarse con el valor de L (si ya ha sido definida). Si L global no existe, debe crearse, y pasarle la lista de referencias de L local.

Semántica de bloques

Sean las reglas

```
<Instr> ::= do <id> := <Expr> S1
            , <Expr> S2
            <CD1> <LInstr> end S5
<CD1>   ::= , <Expr> S3 | ^ S4
```

Semántica asociada al análisis de "do x:=n1,n2,n3; I1; I2; end":

- S1:
- Generar cuádruplas asociadas a instrucción de asignación x:=n1.
- Guardar i=número de cuádrupla siguiente.
- S2:
- Guardar j=número de cuádrupla siguiente.
- Generar cuádrupla (TRG,,x,(n2)).
- Generar cuádrupla (TR,,,).
- S3:
- Generar cuádrupla (+,x,(n3),x).
- Generar cuádrupla (TR,(i),,).
- Hacer cuádrupla[j+1][2]=cuádrupla siguiente.
- S5:
- Generar cuádrupla (TR,(j+2),,).
- Hacer cuádrupla[j][2]=cuádrupla siguiente.

Además, S4:

```
Generar cuádrupla (:=,x,1,x).
Generar cuádrupla (TR,(i),,).
Hacer cuádrupla[j+1][2]=cuádrupla siguiente.
```

Evaluación óptima de las expresiones booleanas

Las operaciones booleanas usualmente se definen así:

```
O | T  F   Y | T  F   NO| T  F
--|-----   --|-----   --|-----
T | T  T   T | T  F     | F  T
F | T  F   F | F  F
```

y la sintaxis adecuada para que la precedencia sea: NO, Y, O. Sin embargo, es posible simplificarlas considerablemente. Basta fijarse en la expresión

```
a Y (b O NO c)
```

Si a es falso, no hace falta calcular el paréntesis, sabemos que el resultado es falso. Por tanto, redefiniremos la sintaxis y la semántica así:

```
<ZB> ::= <EB>
<EB> ::= <TB> O <EB> | <TB>
<TB> ::= <FB> Y <TB> | <FB>
<FB> ::= i | ( <EB> ) | NO <FB>

a O b <=> if (a) TRUE else b;
a Y b <=> if (a) b else FALSE;
NO a  <=> if (a) FALSE else TRUE;
```

Análisis top-down

El programa queda:

```
void ZB (char *cadena)
{
  int FL=0, TL=0, i;
  cuad (":=", "T", NULL, x);
  i = EB (cadena, 0, &FL, &TL);
  FILL (FL);
  cuad (":=", "F", 0, x);
  FILL (TL);
}

unsigned int EB (char *cadena, unsigned int i, int *FL, int *TL)
{
  int tl=0;
  i = TB (cadena, i, FL, &tl);
  MERGE (TL, tl);
  if (i<strlen(cadena)) switch (cadena[i]) {
    case 'O':
      i++;
      FILL (*FL);
      *FL = 0;
      i = EB (cadena, i, FL, TL);
      break;
    default:
      break;
      }
  return i;
}
```

```
unsigned int TB (char *cadena, unsigned int i, int *FL, int *TL)
{
  int fl=0;
  i = FB (cadena, i, &fl, TL);
  MERGE (FL, fl);
  if (i<strlen(cadena)) switch (cadena[i]) {
    case 'Y':
      i++;
      FILL (*TL);
      *TL = 0;
      i = TB (cadena, i, FL, TL);
      break;
    default:
      break;
      }
  return i;
}

unsigned int FB (char *cadena, unsigned int i, int *FL, int *TL)
{
  if (i<strlen(cadena)) switch (cadena[i]) {
    case 'i':
      i++;
      *TL = cuádrupla siguiente;
      *FL = - cuádrupla siguiente;
      cuad (TRB, id, 0, 0);
      break;
    case '(':
      i++;
      i = EB (cadena, i, FL, TL);
      i = C (cadena, i);
      break;
    case 'NO':
      i++;
      i = FB (cadena, i, FL, TL);
      *FL <-> *TL
      break;
    default: error (cadena, i);
      }
  else error (cadena, i);
  return i;
}

unsigned int C (char *cadena, unsigned int i)
{
  if (i<strlen(cadena)) switch (cadena[i]) {
    case ')':
      i++;
      break;
    default: error (cadena, i);
      }
  else error (cadena, i);
  return i;
}

void FILL (int lista)
{
  int i,j,n;
  n = lista;
  while (n) {
    if (n>0) i=2; else i=3;
    j = abs (n);
    n = cuad[j][i];
    cuad[j][i] = cuádrupla siguiente;
    }
}
```

```
void MERGE (int *uno, int dos)
{
  int i,k;
  if (*uno==0) *uno = dos;
  else {
    k = *uno;
    for (;;) {
      if (k>0) i=2; else i=3;
      k = abs (k);
      if (cuad[k][i]==0) break;
      k = quad[k][i];
      }
    cuad[k][i] = dos;
    }
}
```

Analicemos "a O b O NO c"

```
ZB ("a O b O NO c")
  cuad (":=", "T", NULL, "X");
  i = EB ("a O b O NO c", 0, &FL, &TL);
    i = TB ("a O b O NO c", 0, FL, &tl);
      i = FB ("a O b O NO c", 0, &fl, TL);
        case id:
          i=1;
          *TL = 1;
          fl = -1;
          cuad (TRB, a, 0, 0);
      MERGE (FL, fl);
        FL = -1;
    MERGE (TL, tl);
        TL = 1;
    case 'O':
      i=2;
      FILL (FL);
        n = -1;
        while (n) {
          i = 3;
          j = 1; (abs(FL))
          n = 0; (cuad[1][3])
          cuad[1][3] = 2;
          }
      FL = 0;
      i = EB ("a O b O NO c", 2, FL, TL);
        i = TB ("a O b O NO c", 2, FL, &tl);
          i = FB ("a O b O NO c", 2, &fl, TL);
            case 'i':
              i=3;
              *TL = 2;
              *fl = -2;
              cuad (TRB, b, 0, 0);
          MERGE (FL, fl);
            FL = -2;
        MERGE (TL, tl);
          k = 1;
          for (;;) {
            i = 2;
            k = 1;
            }
            cuad[1][2] = 2;
        case 'O':
          i=4;
          FILL (FL);
            n = -2;
            while (n) {
```

```
              i = 3;
              j = 2;  (abs (n))
              n = 0;  (cuad[2][3])
              cuad[2][3] = 3;
              }
            FL = 0;
            i = EB ("a O b O NO c", 4, FL, TL);
              i = TB ("a O b O NO c", 4, FL, &tl);
                i = FB ("a O b O NO c", 4, &fl, TL);
                  case 'NO':
                    i=5;
                    i = FB ("a O b O NO c", 5, FL, TL);
                      case 'i':
                        i=6;
                        *TL = 3;
                        *FL = -3;
                        cuad (TRB, c, 0, 0);
                    FL <-> TL
                MERGE (FL, fl);
                  FL = 3;
              MERGE (TL, tl);
                k = 1;
                for (;;) {
                  i = 2;
                  k = 1;  (abs (k))
                  k = 2;  (cuad[1][2])
                  }
                cuad[2][2] = -3;
    FILL (FL);
      cuad[3][2] = 4;
    cuad (":=", "F", 0, "X");
    FILL (TL);
      cuad[1][2] = 5;
      cuad[2][2] = 5;
      cuad[3][3] = 5;
```

Cuádruplas obtenidas:

```
  0: (":=", "T",, x);
  1: ("TRB", a, 5, 2);
  2: ("TRB", b, 5, 3);
  3: ("TRB", c, 4, 5);
  4: (":=", "F",, x);
  5:
```

Capítulo 6. Generación de código

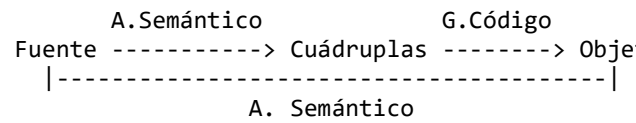
El código generado por un compilador puede ser de uno de los tipos siguientes:

- Código simbólico (*.ASM, hay que pasárselo a un ensamblador).
- Código relocalizable (*.OBJ, hay que pasárselo a un "linker"). Es el sistema más flexible y el más corriente en compiladores comerciales.
- Código absoluto (*.EXE, más eficiente pero menos flexible. Sólo en compiladores antiguos).

Usaremos como ejemplo la traducción a código simbólico (más legible).

Existen dos formas de realizar la generación de código:

- En un solo paso: integrada con el análisis semántico.
- En dos o más pasos: el analizador semántico genera un código intermedio (cuádruplas, notación sufija), a partir del cuál se realiza la generación del código definitivo como un paso independiente.



La primera posibilidad es más rápida, pero más compleja.

Operandos

En el código objeto, los operandos se representan usualmente por su dirección. Esta se almacena a menudo en forma de una base y un desplazamiento, habiendo un número limitado de zonas de datos, cada una con su base. Pueden utilizarse registros índices. (Habrá variables que describan el contenido de éstos, igual que con el acumulador o los registros de trabajo).

Los operandos temporales pueden no recibir memoria asociada, si su rango de actuación no pasa de un registro acumulador. En caso contrario, dicho rango puede almacenarse en la descripción de la variable para declarar libre el espacio en cuanto la variable no sea necesaria. En máquinas modernas se utiliza la pila de la máquina para los operandos temporales.

Información sobre un operando:

- Nombre (o puntero a la tabla de símbolos)
- Tipo: carácter, entero*2, entero*4, real*4, real*8, registro entero, registro real, constante,
- Dimensión vectorial equivalente
- Dirección base
- Indice de equivalencia
- Offset de equivalencia (para punteros)
- Indice
- Offset (para punteros)

Indexado

Vamos a calcular la dirección de un elemento de un "array".

```
ARRAY x[10;5]
... x[4;3] ...
```

En origen 1, la dirección equivalente es la de x[1;1] (base) más un desplazamiento igual a 3*5+2, si x está almacenado por filas. Si se almacena por columnas (FORTRAN), el desplazamiento es 2*10+3.

En general:

```
ARRAY x[a;b; ... ;c;d]
... x[i;...;j;k;l] ...
```

El desplazamiento en origen 1, con almacenamiento por filas, es:

```
(l-1)+(k-1)*d+(j-1)*c*d+...+(i-1)*b*...*c*d
```

El desplazamiento en origen 0, con almacenamiento por filas, es:

```
l+k*d+j*c*d+...+i*b*...*c*d
```

Con almacenamiento por columnas, el orden se invierte.

Más en general, si tenemos

```
ARRAY x[a : b ; c : d]
... x[i;j] ...
```

el desplazamiento es:

```
(i-a)*(d-c+1)+(j-c)
ARRAY x[a : b ; c : d ; e : f]
... x[i;j;k] ...
```

el desplazamiento es:

```
(i-a)*(d-c+1)*(f-e+1)+(j-c)*(f-e+1)+(k-e)
```

Conversiones de tipo

Sea la expresión

```
real A;
int I,J;
...
J = A*I+J
```

Si se pasa por cuádruplas, el análisis semántico podría generar las siguientes:

```
(*,A,I,T1)
(+,T1,J,T2)
(=,T2,,J)
```

En este caso, las rutinas de generación de código tendrán que ocuparse de la compatibilidad y conversión de tipos. También podría complicarse el análisis semántico para que se generasen cuádruplas de conversión:

```
(CVIR,I,,T1)
(*R,A,T1,T2)
(CVIR,J,,T3)
(+R,T2,T3,T4)
(CVRI,T4,,T5)
(=I,T5,,J)
```

En este caso hay más rutinas de generación de código, pero son más sencillas.

Manejo de registros

Si el ordenador tiene un acumulador, tendremos una rutina para cargar un objeto escalar en el acumulador. Habrá una variable (ej. AC) que indique lo que hay en el acumulador en un momento dado. La rutina de carga del acumulador podría ser:

```
int CAC (opd *x, opd *y)
{
  if (AC!=NULL && AC==y) return 1;
  if (AC!=x) {
    if (AC!=NULL) GEN ("MOV ",AC,",AC");
    GEN ("MOV AC,",x);
    AC=x;
    }
return 0;
}
```

Esta rutina puede llamarse así:

```
CAC (x,y)          // Podemos cargar x o y
CAC (x,NULL)       // Tenemos que cargar x
```

y devuelve 0 si x está en el acumulador, 1 si es y.

Si en vez de un acumulador hay un conjunto de registros (ej. AX, BX, CX, DX, SI, DI) tendremos un vector de variables REG (equivalente a AC) y la rutina de carga devolverá el registro seleccionado.

Será preciso distinguir los registros en punto fijo de los de punto flotante. También, algunos registros pueden estar reservados para uso interno del compilador (ej., para bucles, índices, etc). El contenido puede ser una variable o un conjunto de variables (véase la sección sobre operandos).

Una rutina general de carga de registros deberá comenzar por seleccionar el registro a utilizar entre los disponibles. Para ello hay que tener en cuenta el tipo del registro sobre el que se desea cargar (registros enteros o en punto flotante). Si no hay ninguno del tipo deseado, se elegirá uno de los ocupados, perdiendo la información que contiene (si ya no es necesaria) o guardándola en la posición de memoria asociada. Una vez seleccionado el (los) registro(s), la carga propiamente dicha dependerá del tipo del objeto a cargar. Es conveniente hacer una tabla en función de dicho tipo, como la siguiente:

```
  Carga de X:
Tipo                          Tipo de X
reg    char        short      int-r   const      real    real-r
---- ------------------------------------------------------------
int  XOR RH,RH     MOV RX,X     -     MOV RX,X   FLD X   FISTP T
     MOV RL,X                                   (Carga) (Carga)
real XOR RH,RH     FILD X     MOV T,X MOV T,X    FLD X      -
     MOV RL,X                 (Carga) (Carga)
```

(Carga)

Expresiones

La construcción de tablas de código generado es muy útil también para las operaciones que aparecen en las expresiones, especialmente para las diádicas, donde la tabla será de doble entrada (tipo del argumento izquierdo, tipo del derecho). A menudo, el cambio de tipo se puede realizar de forma más o menos directa a través de la operación de carga en registro definida anteriormente.

Veamos como ejemplo la tabla correspondiente a la operación suma:

```
  X+Y:
Tipo                      Tipo de Y
de X    char      short    int-r    const    real     real-r
------  ------------------------------------------------------
char    CARGA X   X<->Y    X<->Y    CARGA X  CARGA X  CARGA X
short   CARGA Y   CARGA Y  ADD Y,X  CARGA Y  CARGA X  FIADD X
int-r   CARGA Y   ADD X,Y  ADD X,Y  ADD X,Y  MOV T,X  MOV T,X
                                             (Suma)   (Suma)
const   X<->Y     X<->Y    X<->Y       -     CARGA X  FADD X
real    CARGA Y   CARGA Y  X<->Y    X<->Y    CARGA Y  FADD X
real-r  X<->Y     X<->Y    X<->Y    X<->Y    X<->Y    FADD Y
```

En la tabla anterior hemos aprovechado el hecho de que la operación es conmutativa y algunas casillas pueden reducirse a otras sin más que cambiar el orden de los operandos. La utilización de la operación CARGA reduce aún más la parte de la tabla verdaderamente dedicada a la generación del código. Siempre que realizamos una de estas operaciones, se entiende que hay que volver a aplicar la tabla, indexándola por los tipos de los nuevos operandos.

Programación de la tabla de la suma en seudocódigo:

```
    Label Tabla[n][n] = {LCX, LXG, LXG, LCX, LCX, LCX}
                        {LCY, LCY, L1,  LCY, LCX, L2}
                        {LCY, L3,  L3,  L3,  L4,  L4}
                        {LXG, LXG, LXG, 0,   LCX, L5}
                        {LCY, LCY, LXG, LXG, LCY, L5}
                        {LXG, LXG, LXG, LXG, LXG, L6}
  L: GOTO Tabla[tipox][tipoy];
  LCX: CARGA X;
       GOTO L;
  LCY: CARGA Y;
       GOTO L;
  LXG: X<->Y;
       GOTO L;
  L1:  GEN (ADD Y,X);
       return;
  L2:  GEN (FIADD Y);
       return;
  L3:  GEN (ADD X,Y);
       return;
  L4:  GEN (MOV T,X);
       GOTO L;
  L5:  GEN (FADD X);
       return;
  L6:  GEN (FADD Y);
```

```
      return;
```

En una operación no conmutativa (como la resta) el número de casillas generadoras de código aumenta. En las operaciones monádicas la tabla se reduce normalmente a una tabla de entrada simple. Por ejemplo, para el cambio de signo:

```
  -Y:
                        Tipo de Y
       char      short    int-r    const    real     real-r
       ---------------------------------------------------
char   CARGA Y   CARGA Y  NEG Y      -      CARGA Y  FCHS
```

Generación a partir de cuádruplas

Para funciones diádicas conmutativas (suma, producto):

```
    SUMA (opd *x, opd *y, opd *z)
    {
      if (CAC (x, y)) GEN ("ADD AC,",x);
      else GEN ("ADD AC,",y);
      AC=z;
    }
```

(+, O1, O2, R) se traduce por SUMA (O1, O2, R).

La multiplicación se haría igual, sustituyendo "ADD" por "MUL".

Para funciones diádicas no conmutativas:

```
    RESTA (opd *x, opd *y, opd *z)
    {
      CAC (x, NULL);
      GEN ("SUB AC,",y);
      AC=z;
    }
```

(-, O1, O2, R) se traduce por RESTA (O1, O2, R).

La división se haría igual, sustituyendo "SUB" por "DIV".

Para funciones monádicas:

```
    NEG (opd *x, opd *z)
    {
      CAC (x, NULL);
      GEN ("NEG AC");
      AC=z;
    }
```

(@, O1,, R) se traduce por NEG (O1, R).

Ejemplo: sea la expresión A*((A*B+C)-C*D)

Las cuádruplas son:

```
  Cuádrupla     Se genera       Valor de AC
  ------------  ----------      -----------
```

```
(*,A,B,T1)    MOV AC,A          A
              MUL AC,B          T1
(+,T1,C,T2)   ADD AC,C          T2
(*,C,D,T3)    MOV T2,AC
              MOV AC,C          C
              MUL AC,D          T3
(-,T2,T3,T4)  MOV T3,AC
              MOV AC,T2         T2
              SUB AC,T3         T4
(*,A,T4,T5)   MUL AC,A          T5
```

A partir de notación polaca

Es el mismo algorimo que se explicó en el capítulo anterior para la evaluación de expresiones en notación polaca, sustituyendo las evaluaciones (intérprete) por generación de código (compilador).

Si el acumulador está en segundo lugar en la pila, al introducir un operando nuevo en la pila hay que generar una instrucción que pase el acumulador a una variable intermedia y sustituir el acumulador por el nombre de ésta. Con ello, un operador diádico no tendría que preocuparse por el acumulador. Uno monádico tendría que comprobar sólo si es el segundo operando de la pila.

Generación de código para la asignación

Sea la cuádrupla

```
(=,B,,A)
```

En general, habrá que generar

```
MOV AC,B
MOV A,AC
```

En algunas máquinas existe una instrucción que permite pasar de memoria a memoria, tal como

```
MOV A,B
```

También puede ser que, al generar esta instruccción, nos encontremos con que el acumulador contiene B (AC="B"), en cuyo caso bastaría con generar

```
MOV A,AC
```

La asignación puede llevar implícitas diversas conversiones de tipo, que pueden haber sido tratadas previamente por el analizador semántico.

También puede haber conversión implícita de la profundidad de los punteros, como en

```
ref int A;
int B;
...
B := A;
```

En este caso, el código generado debería ser:

```
MOV índice,A
MOV AC,[índice]
MOV B,AC
```

Veamos un ejemplo de tabla de conversión de tipos para la asignación:

```
  X=Y:
Tipo                    Tipo de Y
de X    char      short    int-r    const    real     real-r
------  ------------------------------------------------------
char    CARGA Y   CARGA Y  MOV X,YL MOV X,Y  CARGA Y  FISTP T
                                                     MOV R,T
                                                     MOV X,RL
short   CARGA Y   CARGA Y  MOV X,Y  MOV X,Y  CARGA Y  FISTP X
int-r   CARGA Y*  CARGA Y* MOV X,Y* MOV X,Y  CARGA Y  FISTP T
                                                     MOV X,T
const     -         -        -        -        -        -
real    CARGA Y   CARGA Y  MOV T,Y  CARGA Y  CARGA Y  FSTP X
                           CARGA T
real-r  CARGA Y*  CARGA Y* MOV T,Y  CARGA Y* CARGA Y*   -
                           CARGA T*
```

Generación de código para GOTO

Ver analizador semántico (se mantiene lista encadenada hasta que aparece la etiqueta).

Información sobre una etiqueta:

- Nombre (o puntero a la tabla de símbolos)
- Tipo: interna, externa, subrutina, función, corrutina...
- Si está definida o no.
- Dirección.

Generación de código para instrucciones condicionales

- if (expr_b) inst1;
- ```
    expr_b
    TRF n
    inst1
  n:
  ```
- if (expr_b) inst1; else inst2;
- ```
    expr_b
    TRF m
    inst1
    TR n
  m: inst2
  n:
  ```

Generación de código para bucles

- for (inst_i; expr_b; fin) inst;
- ```
    inst_i       inst_i
  m: expr_b    p: expr_b
    TRF n        TRF s
  ```
  ```

```
     inst          TR r
     fin       q: fin
     TRT m         TR p
  n:           r: inst
                  TR q
               s:
```

```
while (expr_b) inst;
```

```
 m: expr_b
    TRF n
    inst
    TR m
 n:
```

```
do inst; while (expr_b);
```

```
 m: inst
    expr_b
    TRT m
```