Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Generación de Código Intermedio

Lenguajes de Programación y Procesadores de Lenguajes

V. 22

Tabla de contenido

1. Códigos intermedios

Además de analizar el código fuente (fase de análisis) un compilador debe generar código (fase de síntesis). Esta generación de código se realiza en dos fases. Una primera fase en la que el "front-ent" del compilador produce como resultado una *representación intermedia* ("Intermediate Representation" o IR) del programa fuente y una segunda fase en la que el "back-end" producirá el código objeto.

La fase de síntesis puede generar directamente como salida un código máquina, pero en ese caso el compilador será totalmente dependiente de la máquina destino. Además, debido a la gran disparidad entre el tipo de instrucciones que se emplean en los lenguajes de alto nivel y en los lenguajes máquina, la construcción del generador de código será larga y el código resultante difícil de mantener. Otra alternativa consiste en que la fase de síntesis genere código en otro lenguaje de alto nivel. En este caso se requiere del empleo de otro compilador para la máquina destino que finalmente genere el código máquina. La construcción del traductor será mucho más sencilla, pero a cambio el código ejecutable que obtendremos consumirá en general más recursos (tiempo y memoria).

La utilización de una representación intermedia (IR) aparece como una solución a mitad de camino entre las dos anteriores. El proceso de generación de código queda dividido en dos etapas de traducción: una primera fase independiente de la máquina destino en la que se genera una representación intermedia, y una segunda fase en la que, a partir de la representación intermedia, se genera el código objeto ejecutable para la máquina destino. En la primera etapa conocida como generación de código intermedio se genera una representación intermedia del código fuente que se va a compilar. Esta representación intermedia, independiente de la máquina para la que se generará código y del lenguaje fuente, será la entrada del módulo generador de código encargado de realizar la segunda etapa de traducción. Esta tercera opción es la que vamos a estudiar con más detalle a lo largo de este capítulo.

Un código intermedio es una estructura de código cuya complejidad está entre el código fuente en el lenguaje de alto nivel y el código máquina. Este código intermedio puede ser considerado como la interfaz entre el generador de código ("back-end") y las fases anteriores del compilador ("front-end"). Su utilización presenta algunas ventajas frente a la generación directa de un código máquina:

- Se facilita la reutilización de toda la etapa inicial del compilador ("front-end") para la construcción de compiladores de un mismo lenguaje fuente para distintas máquinas. (Fig. 2.1), y de toda la etapa final para construcción de compiladores de otros lenguajes fuente sobre la misma plataforma.
- Se puede aplicar a la representación intermedia un optimizador de código independiente de la máquina.

- Facilita la división en fases del proyecto: Permite abstraerse de los detalles propios de la máquina destino en el desarrollo de la etapa inicial del compilador y de los propios del lenguaje fuente en el desarrollo de la etapa final.

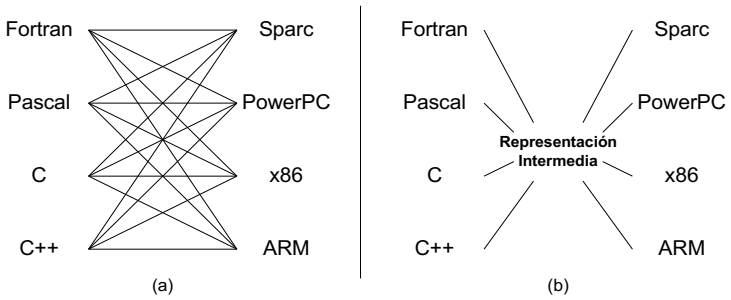


Figura 2.1. Compiladores para 4 lenguajes y 4 máquinas sin (a) y con (b) utilización de un código intermedio.

Una buena representación intermedia debe cumplir las siguientes propiedades:

- Debe ser sencilla de generar y manipular.
- Debe ser adecuada para permitir su traducción al código máquina de todas las máquinas-destino deseadas.
- Cada construcción debe tener un significado claro y simple que facilite la implementación del módulo optimizador.

El código generado por un compilador de Java, conocido como *bytecode*, puede considerarse como un ejemplo de código intermedio con la característica peculiar de que que posteriormente es ejecutado en la máquina virtual.

1.1. Tipos de representaciones intermedias

Hay distintos tipos de representación intermedia. En el capítulo 10 de [Tremblay, 85] puede encontrarse una explicación más extensa las formas de representación intermedias que se presentan a continuación.

Notación postfija

Notación sencilla empleada en máquinas de pila donde un operador se coloca a continuación de sus operandos. Por ejemplo, la instrucción de asignación x := a+b+(c*d) quedaría en notación postfija como: x ab+cd*+ :=

Este tipo de representación intermedia es poco adecuada para la fase de optimización y es difícil de emplear para representar estructuras de control de flujo e instrucciones de selección.

Árbol de sintaxis abstracta

Un *árbol de sintaxis abstracta* (“Abstract Sintax Tree” o AST) no es más que un árbol en el que un nodo interior representa un operador y donde sus operandos serán sus descendientes

directos. En la Fig. 2.2 se puede observar como queda un árbol de sintaxis abstracta para la asignación x:= (a+b*c) / d.

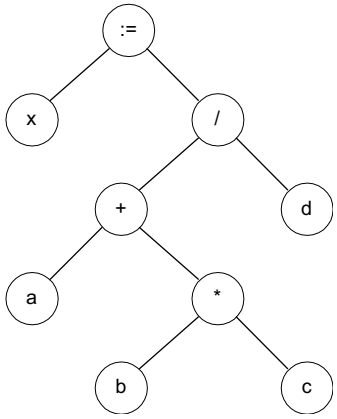


Figura 2.2. Árbol de sintaxis abstracta para la instrucción x:= (a+b*c) / d.

Grafo Dirigido Acíclico (GDA)

Una variante de los árboles de sintaxis abstracta son los *grafos dirigidos acíclicos* (*GDA*). La principal diferencia con los árboles de sintaxis abstracta es que en los primeros se permite que un nodo tenga más de un padre. Este tipo de representación facilita el análisis del código y su optimización, por lo que es muy empleada en compiladores optimizadores.

Código de tres direcciones (cuartetos)

Representación intermedia en la que las instrucciones constan de un operador y tres direcciones: dos para los operandos y una tercera para el resultado. Las instrucciones tendrán la forma genérica

<operador> <op1> <op2> <resultado>

Por ejemplo, la instrucción de asignación x := (a+b*c)/d se podría representar mediante una secuencia de instrucciones en un código de tres direcciones como

```
   Operador  op1  op2    res
(1) MULT     b     c     t1
(2) SUMA     a     t1    t2
(3) DIV      t2    d     t3
(4) ASIGNA   t3          x
```

Un código de tres direcciones no es más que una representación lineal de un árbol de sintaxis abstracta o de un GDA en el que se ha empleado una variable temporal para representar los valores de los nodos interiores.

Forma SSA (Static Single-Assignment)

La forma SSA es una representación intermedia similar al código de tres direcciones en la que todas las asignaciones se realizan a variables con distinto nombre. Este tipo de representación facilita la aplicación de transformaciones que optimizan el código

Código de dos direcciones (tercetos)

En un código de tres direcciones se suele emplear un gran número de variables auxiliares (temporales) para almacenar los resultados parciales durante la evaluación de una expresión. Un código de dos direcciones evita este problema permitiendo que los operandos que aparecen en una instrucción sean apuntadores a las instrucciones cuyo resultado se desea emplear.

La forma general de una instrucción de dos direcciones es

<op> <operando1> <operando2>

El mismo ejemplo de instrucción de asignación que hemos vistos representada mediante un código de tres direcciones, x := (a+b*c)/d se podría representar mediante un código de dos direcciones como

```
Operador     op1  op2
(1) MULT     b     c
(2) SUMA     a    (1)
(3) DIV     (2)   d
(4) ASIGNA  (3)   x
```

Donde (1), (2) y (3) representan punteros a las instrucciones de código dos direcciones 1, 2 3 respectivamente.

La principal ventaja del código de dos direcciones frente al de tres es el ahorro de variables temporales.

1.2. Instrucciones del código de tres direcciones

Para este capítulo hemos escogido como representación intermedia un código de tres direcciones. Se muestran a continuación los tipos de instrucciones de este código y la forma general que tendrán. En la mayoría de los casos, los operadores podrán ser constantes, direcciones de memoria donde se encuentra el valor a usar, o directamente direcciones de memoria.

Generación de etiquetas

Los destinos de los saltos se indican mediante etiquetas. Al definir el código intermedio podemos elegir entre dos formas de generar estas etiquetas:

Generación explícita de etiquetas: Cada vez que se desee poner una etiqueta a una instrucción de código intermedio se debe crear y emitir una etiqueta que tendrá la forma:

<nombre_etiqueta> **:**

Esta forma es la usada habitualmente por los lenguajes ensambladores**.** En el ejercicio 9.20 se puede ver un ejemplo de utilización de este tipo de etiquetas.

Generación implícita de etiquetas: Podemos suponer que cada instrucción de código intermedio generada tendrá implícitamente una etiqueta que se corresponderá con un número entero. Este número se corresponde con el número de orden de la instrucción dentro de programa. Esta forma es empleada en algunos lenguajes de alto nivel como Basic, que comienza cada línea con un número que representa la etiqueta de la instrucción.

El código intermedio que se emplea en este capítulo utiliza etiquetas implícitas.

1.3. Generación de código intermedio mediante una gramática de atributos

En temas anteriores se ha mostrado la utilización de las gramáticas atribuidas como formalismo para la especificación semántica. En este tema se seguirán empleando las gramáticas atribuidas, pero sus acciones generan código intermedio de tres direcciones. Para ello vamos a emplear un procedimiento llamado *emite* que recibirá como parámetros una instrucción de código intermedio y sus argumentos. Este procedimiento se encargará de almacenar secuencialmente las instrucciones en una estructura intermedia (que podría ser un fichero o un vector de instrucciones de código intermedio). De esta forma, como resultado colateral de la evaluación de la gramática, en la estructura intermedia empleada por *emite*, tendremos toda la secuencia de instrucciones de código intermedio generadas, las cuales forman el programa en código intermedio equivalente al programa fuente. Este programa en código intermedio tendrá todas sus instrucciones etiquetadas con un número entero: el número de instrucción que será generado automáticamente por el procedimiento *emite*.

Las expresiones que aparezcan entrecomilladas en la llamada al procedimiento *emite*, se tomarán literalmente, mientras que el resto de expresiones se evaluarán antes de generar la instrucción de código intermedio.

Ejemplo 2.1.-

emite (x ':=' 5 '+' 9)
emite (x ':=' 5 + 9)

La primera llamada al procedimiento *emite* generará la instrucción de código tres direcciones correspondiente a la asignación en la posición x del resultado de sumar las constantes 5 y 9 (se emite una instrucción de suma, es decir, la suma se realizará en tiempo de ejecución). En cambio, la segunda llamada al procedimiento *emite* generará una instrucción que almacenará en la posición x el valor 14 (se emite una instrucción de asignación, la suma se ha realizado en tiempo de compilación).

En las gramáticas atribuidas que construyamos para generar código intermedio utilizaremos la siguiente notación:

id.nom: Atributo asociado el símbolo léxico identificador que representa su lexema (nombre). Su valor es devuelto por el analizador léxico. Este nombre nos permitirá buscar la posición en la tabla de símbolos (TDS) donde se encuentra la información del identificador.

id.pos: Atributo del símbolo léxico que representa la posición de memoria que contendrá durante la ejecución el valor de la variable representada por el identificador. Se le asigna a la variable cuando se declara, se almacena en la TDS y puede ser consultado mediante la función *BuscaPos*(...). En un lenguaje con estructura de bloques esta posición suele estar representada por el par (nivel, desplazamiento). Cuando otros símbolos de la gramática tengan un atributo con el mismo (.pos) nombre tendrán el mismo significado.

id.tipo: Atributo que representará la expresión de tipo asignada al identificador. Está almacenado en la TDS y se puede consultar mediante la función *BuscaTipo*(...). Cuando otros símbolos de la gramática tengan un atributo con el mismo nombre tendrán el mismo significado.

num.val: Atributo asociado al símbolo léxico constante numérica (num) que representa su valor. Es devuelto por el analizador léxico. Cuando otros símbolos de la gramática tengan un atributo con el mismo nombre tendrán el mismo significado.

Para simplificar los esquemas de traducción dirigidos por la sintaxis, nos vamos a permitir emplear en las acciones semánticas variables globales. Estas variables globales podrán ser accedidas desde cualquier acción semántica, es decir, se corresponderán con variables globales del código fuente del compilador desarrollado. Para distinguirlas, aparecerán siempre en mayúsculas. Si en alguna acción semántica aparece una variable en minúsculas se tratará de una variable local asociada al símbolo no-terminal del lado izquierdo de la producción en la que aparece.

SIGINST: Variable global del compilador que estamos construyendo que representa el número de la SIGuiente INSTrucción que se va a generar. Esta variable es inicializada a 0 y el procedimiento *emite* se encarga de incrementarla automáticamente cada vez que se emite una instrucción.

Dada la siguiente gramática que representa una instrucción de asignación de un lenguaje de alto nivel similar a C

Asig → **id** = E

E → E + E

E → **id**

podemos construir un esquema de traducción dirigido por la sintaxis (ETDS) que genere código de tres direcciones para instrucciones de asignación usando el procedimiento *emite*:

Asig → **id** = E	{ Asig.pos := BuscaPos (id.nom) ; emite (Asig.pos ':=' E.pos); }
E → E_1 + E_2	{ E.pos := CrearVarTemp() ; emite (E.pos ':=' E_1.pos '+' E_2.pos) }
E → **id**	{ E.pos := BuscaPos (id.nom) }

Veamos cada una de las acciones semánticas asociadas a estas producciones.

Asig → **id** = E { Asig.pos := BuscaPos (id.nom) ;
emite (Asig.pos ':=' E.pos) }

En el ETDS (esquema de traducción dirigido por la sintaxis) anterior se puede observar que no aparecen acciones semánticas necesarias para realizar las comprobaciones semánticas. Esto es debido a que este capítulo se centra únicamente en la generación de código intermedio.

Se puede observar como en la primera acción semántica se realiza una llamada a la función *BuscaPos*. La función *BuscaPos* recibe como argumento el lexema (nombre) del identificador, busca la entrada del identificador en la TDS y devuelve la posición de memoria que se ha asignado a dicho identificador. Esta posición de memoria en un lenguaje con estructura de bloques será habitualmente el nivel de anidamiento (o contexto) del identificador y su desplazamiento relativo.

La siguiente acción semántica es una llamada al procedimiento *emite*. Éste emitirá una instrucción en código de tres direcciones que asignará en la posición del identificador id el contenido de la posición de memoria representada por E.pos. También se puede observar que la acción semántica devuelve en el atributo sintetizado Asig.pos la posición de memoria donde estará en tiempo de ejecución el valor de la variable id.nom. Esto se hace porque en C una

instrucción de asignación debe devolver el valor asignado a la variable del lado izquierdo de la asignación. La utilidad de este atributo sintetizado se puede observar si añadimos a la gramática anterior la producción

$$E \rightarrow Asig \qquad \{\ E.pos := Asig.pos\ \}$$

donde se puede observar que una expresión puede ser una asignación.

La siguiente producción del ETDS es:

$$E \rightarrow E_1 + E_2 \qquad \{\ E.pos := CrearVarTemp()\ ;\ emite\ (\ E.pos\ ':='\ E_1.pos\ '+'\ E_2.pos\ )\ \}$$

La primera acción de esta producción realiza una llamada a la función *CrearVarTemp()*. Está función devolverá la dirección de la primera posición de memoria libre (no asignada) dentro del área de datos del bloque que se está compilando. En esa posición de memoria se almacenará el valor de una variable temporal (de ahí el nombre de la función).

La segunda acción emite una instrucción de código de tres direcciones. Esta instrucción se almacenará en la estructura intermedia empleada por *emite* para guardar el código intermedio. Cuando en tiempo de ejecución se ejecute la instrucción, se asignará a la posición de memoria representada por E.pos (la posición que devolvió CrearVarTemp) el resultado de sumar el contenido de las posiciones de memoria representadas por E_1.pos y E_2.pos. Podemos observar que E.pos es un atributo sintetizado que *toma valor* en esta producción mientras que E_1.pos y E_2.pos son dos atributos sintetizados cuyo valor se *usa* en esta producción.

$$E \rightarrow \mathbf{id} \qquad \{\ E.pos := BuscaPos\ (id.nom\ )\ \}$$

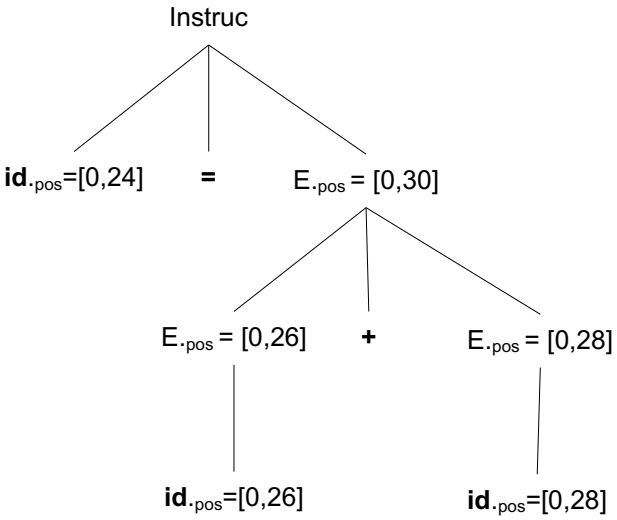
La única acción semántica asociada a esta producción se encarga de dar al atributo E.pos la posición de memoria asignada al identificador id. Para ello se realiza una llamada a la función *BuscaPos* que recibe como argumento el lexema del identificador (id.nom), lo busca en la tabla de símbolos y devuelve la posición de memoria que se le adjudicó en la fase de asignación de memoria.

Ejemplo:

Veamos el comportamiento de este ETDS mediante la construcción del árbol anotado correspondiente a la cadena a = b + c (Fig. 2.3). Suponemos que en la TDS tenemos la información asociada a cada identificador que aparece en la Fig. 2.3.

TDS

Nom	tipo	posición [nivel, desp]
a	tentero	[0,24]
b	tentero	[0,26]
c	tentero	[0,28]



Código tres direcciones generado

```
(1) [0,30] := [0,26] + [0,28]
(2) [0,24] := [0,30]
```

Figura 2.3. Árbol anotado y código intermedio generado para la cadena a=b+c.

Para que resulte más cómodo de leer el código intermedio, en lugar de poner las posiciones de memoria, en el resto de este capítulo se utilizará el nombre de la variable a la que corresponde. En el caso de las variables temporales, su nombre comenzará por la letra minúscula t seguida de un número entero que indicará el orden en el que ha sido generada: t1, t2, … En la Fig. 2.3 se puede observar como al llamar a la función CrearVarTemp(), ésta ha devuelto como posición de memoria la primera posición libre dentro del área de datos del bloque para el que se estaba generando código (nivel 0, desplazamiento 30). Como ésta es la primera variable temporal que se crea se le ha dado el nombre t1. De esta forma el código generado anteriormente puede representarse para mayor claridad como:

```
(1) t1 := b + c
(2) a := t1
```

Veamos a continuación como se puede extender el ETDS anterior para generar código también para otras expresiones aritméticas sencillas.

$E \rightarrow E_1$ * E_2	{ E.pos := CrearVarTemp() ; emite (E.pos ':=' E_1.pos '*' E_2.pos) }
$E \rightarrow$ **num**	{ E.pos := CrearVarTemp() ; emite (E.pos ':=' num.val) }
$E \rightarrow$ **(** E_1 **)**	{ E.pos := E_1.pos }
$E \rightarrow$ - E_1	{ E.pos := CrearVarTemp() ; emite (E.pos ':= -' E_1.pos) }

Las acciones semánticas de la primera producción son semejantes a las que aparecían en la producción que representaba la suma de expresiones, pero en este caso se genera una instrucción de código intermedio de multiplicación. En la segunda producción podemos observar que cuando la expresión es una constante numérica se ha decidido crear una variable temporal y generar una instrucción de código intermedio que asigne en la posición de la variable temporal el valor de la constante numérica (num.val) que proporciona el analizador léxico. Una opción más adecuada habría sido que el no-terminal E devolviese el valor de la constante numérica, lo que habría evitado la creación de una variable temporal. Sin embargo, esta segunda opción habría obligado a que el no-terminal E, además del atributo .pos tuviese al menos otro atributo para devolver valores constantes, lo que obligaría a añadir más acciones semánticas a todas las producciones del no-terminal E. Para simplificar los esquemas de traducción hemos elegido la primera opción. Hay que tener en cuenta que esta elección no afectará a la calidad del código finalmente generado porque este tipo de variables temporales son fácilmente eliminadas durante la fase de optimización de código intermedio.

En la tercera producción, una expresión (E) se reescribe como una expresión entre paréntesis (E_1). Puesto que el valor de una expresión entre paréntesis es el mismo de la expresión sin paréntesis, basta con hacer que el atributo E.pos tome el mismo valor del atributo E_1.pos. De esta manera los dos atributos representan la misma posición de memoria (y por lo tanto el mismo valor de la expresión).

Finalmente, en la última expresión observamos que E se reescribe como una expresión cambiada de signo. Este caso es diferente del anterior, puesto que la expresión representada por E no tomará el mismo valor que E_1, por lo tanto, es necesario utilizar una nueva posición de memoria para almacenar el nuevo valor.

Ejercicio

> Construye una gramática no-ambigua equivalente a la anterior. A continuación, asocia a cada producción las acciones semánticas necesarias para generar el código intermedio correspondiente.

2. Acceso a miembros de una estructura y elementos de una matriz

2.1. Acceso a miembros de una estructura

El acceso a los miembros de una estructura en C depende de la información que se almacene en la TDS sobre la posición de los miembros. La forma habitual consiste en almacenar, junto a cada miembro, su desplazamiento con respecto al comienzo del área de memoria asignada a esa estructura (Fig. 2.4).

Ejemplo 2.2.- Dada la definición de las variables a y b de tipo estructura en el lenguaje C

```
struct ej {
          int c1 ;
          float c2 ;
        }a, b;
```

TALLA_REAL = 4
TALLA_ENTERO = 2

La TDS quedaría:

TDS

nom	tipo	posición	…	ref
a	testructura	0 , 124		
b	testructura	0 , 130		

Tabla de campos

nom	tipo	posición
c1	tinteger	0
c2	treal	2

Figura 2.4. Asignación de memoria en la TDS para los miembros de una estructura.

De esta forma, en la entrada en la TDS de cada miembro de un tipo de estructura concreta se almacenará el mismo desplazamiento independientemente de que hayan declaradas más de una variable de ese tipo. Lo único que cambiará será la dirección base de cada variable de ese tipo de estructura. Evidentemente antes de generar código intermedio para acceder a un miembro de una estructura será necesario sumar a la dirección base de la variable estructura, el desplazamiento asignado al miembro. En la Fig. 2.5 se muestra cómo quedaría un ETDS para acceder a los miembros de una estructura de esta manera.

Se ha usado la función *EsMiembro(id_1.nom, id_2.nom)* que devolverá el valor TRUE si el nombre id_2.nom es un miembro de la estructura de nombre id_1.nom. La función *BuscaPos(id_1.nom)* devuelve la posición de memoria base a partir de la cual se encontrarán en tiempo de ejecución los miembros de la estructura id_1.nom. La llamada a *BuscaPosMiembro(id_1.nom, id_2.nom)* devolverá el desplazamiento relativo del miembro a partir de la posición de memoria base de la estructura.

$E \rightarrow id_1.id_2$	{ si BuscaTipo (id_1.nom) <> testructura ent MemError(); sino base := BuscaPos (id_1.nom) ; si no EsMiembro (id_1.nom, id_2.nom) ent MemError() sino desp_miembro:=BuscaPosMiembro(id_1.nom, id_2.nom); E.pos := base + desp_miembro ; }
$Asig \rightarrow id_1.id_2 = E$	{ si BuscaTipo (id_1.nom) <> testructura ent MemError(); sino base := BuscaPos (id_1.nom) ; si no EsMiembro (id_1.nom, id_2.nom) ent MemError() sino desp_miembro:=BuscaPosMiembro(id_1.nom, id_2.nom); Asig,pos := base + desp_miembro ; emite(Asig.pos ':=' E.pos); }

Figura 2.5. ETDS Para el acceso a los miembros de una estructura.

2.2. Acceso a elementos de un array

En los lenguajes de alto nivel podemos ver que frecuentemente se permite definir matrices (arrays) de varias dimensiones, con un límite inferior y superior (Pascal) o indicando el número de elementos (C) de cada una de sus dimensiones. En cambio, en el código intermedio definido solo disponemos de instrucciones de asignación con un modo de direccionamiento relativo: de `z:=a[x]` (en la posición `z` se almacena el valor de la posición resultante de sumar a la dirección base `a` el contenido de la posición `x`) y `a[x]:=z`. Esto obliga a realizar proceso de traducción que requiere el cálculo de la posición de memoria que ocupará en tiempo de ejecución cada elemento del array definido en el programa fuente.

2.2.1. Acceso a elementos de un array multidimensional con límites inferior y superior. Pascal

Una forma bastante frecuente y cómoda de almacenar las matrices consiste en almacenar todos sus elementos en posiciones consecutivas de memoria. Con esta representación, cada vez que nos encontremos en el lenguaje de alto nivel con una expresión donde aparezca un elemento de una matriz, será necesario calcular el desplazamiento que hay que sumar a la dirección base de la matriz para acceder al elemento indicado. Veamos como se realiza este cálculo en el caso en el que se pueden definir límites inferiores y superiores, como ocurre en el lenguaje Pascal.

Si suponemos la declaración de una variable A de tipo vector (matriz de una dimensión) en Pascal:

```
A: array [inf..sup] of integer ;
```

y se desea acceder al elemento `A[i]`, éste se encontrará en la posición

$$base + (i - inf) * Talla$$

donde

base es la dirección base donde se encuentra el primer elemento del vector,

inf es el límite inferior de vector, y

Talla es el número deposiciones de memoria que ocupa cada elemento del vector (dependiente del tipo de elementos que contiene el vector).

De igual forma, si se define una matriz de dos dimensiones:

`A: array [inf1..sup1, inf2..sup2] of integer ;`

la fórmula para acceder a un elemento `A[i1,i2]` sería:

$$base + ((i_1 - inf_1) * n_2 + (i_2 - inf_2)) * Talla$$

donde n_2 es el número de elementos de la segunda dimensión de la matriz ($n_2 = sup_2 - inf_2 + 1$)

que puede quedar como:

$$base + ((i_1 * n_2 + i_2) - (inf_1 * n_2 + inf_2)) * Talla$$

$(inf_1 * n_2 + inf_2)$ puede ser calculado en tiempo de compilación y recibe el nombre de *desplazamiento inicial* de la matriz. Este valor, puesto que no depende del elemento de la matriz al que se quiera acceder, puede ser almacenado en la TDS. En cambio, $(i_1 * n_2 + i_2)$ tendrá que ser calculado en tiempo de ejecución y por lo tanto habrá que generar código para calcularlo.

En general, dada la declaración de una matriz n-dimensional:

`A: array [inf1..sup1, inf2..sup2, ... infn..supn] of integer ;`

la fórmula para acceder a un elemento `A[i1,i2,...in ]` quedará como:

$$base + (((((i_1 * n_2 + i_2) * n_3 + i_3 \ldots) * n_n + i_n) - (((((inf_1 * n_2 + inf_2) * n_3 + inf_3 \ldots) * n_n + inf_n)) * Talla$$

donde $(((((inf_1 * n_2 + inf_2) * n_3 + inf_3 \ldots) * n_n + inf_n)$ es el desplazamiento inicial de la matriz.

Puesto que el acceso a un elemento de una matriz puede aparecer en el lado derecho o izquierdo de una asignación, las producciones que definen la sintaxis de este acceso aparecerán en dos lugares distintos de la gramática. Como el código generado para los dos casos es prácticamente el mismo, se muestra únicamente uno de los casos:

Inst_simple → **id [** Lista_indices **] :=** E

Lista_indices → E Resto_LI

Resto_LI → **,** E Resto_LI

| ε

En la Fig. 2.6 se muestra un ETDS que genera código intermedio para el acceso a los elementos de una matriz n-dimensional en Pascal.

Inst_simple -> **id**	{ Lista_indices.nom := id.nom }
[Lista_indices **] :=**	{ temp := Lista_indices.pos ; emite (temp := temp '–' Desp_inicial(id.nom)) ; emite (temp := temp '*' Talla (id.nom)) }
E	{ pos := BuscaPos (id.nom) ; emite (pos '[' temp ']' := E.pos }
Lista_indices → E	{ temp := CrearVarTemp() emite (temp := E.pos); Resto_LI.ndim := 1; Resto_LI.temp := temp ; Resto_LI.nom := Lista_indices.nom }
Resto_LI	{ Lista_indices.pos := Resto_LI.pos }
Resto_LI → **,** E	{ Resto_LI.ndim := Resto_LI.ndim + 1; emite (Resto_LI.temp := Resto_LI.temp'*'Num_elementos(Resto_LI.nom, Resto_LI.ndim)); emite (Resto_LI.temp := Resto_LI.temp '+' E.pos) $Resto\_LI_1$.ndim := Resto_LI.ndim ; $Resto\_LI_1$.temp := Resto_LI.temp ; $Resto\_LI_1$.nom := Resto_LI.nom ; }
$Resto\_LI_1$	{ Resto_LI.pos:= $Resto\_LI_1$ pos }
Resto_LI → ε	{ Resto_LI.pos:= Resto_LI.pos }

Figura 2.6. ETDS para el acceso a los elementos de un array en Pascal.

La función *Desp_inicial(R_Inst_simple.nom)* devuelve el desplazamiento inicial de la matriz de nombre R_Inst_simple.nom que se guardó en la TDS cuando se declaró la matriz. De igual forma, la función *Talla(R_Inst_simple.nom)*, devuelve el tamaño de cada elemento de la matriz, y *Num_elemen(Lista_indices.nom,n_dim)* devuelve el número de elementos de la dimensión n_dim de la matriz Lista_indices.nom.

Ejemplo 2.3.- Veamos qué código generará este ETDS para el acceso a un elemento de una matriz bidimensional que se define en el siguiente trozo de código fuente en Pascal:

```
var
  ...
   A: array [2..11,1..20] of real ;
   x,y: integer ; z: real;
begin
   ...
   A[x,y] := z
   ...
```

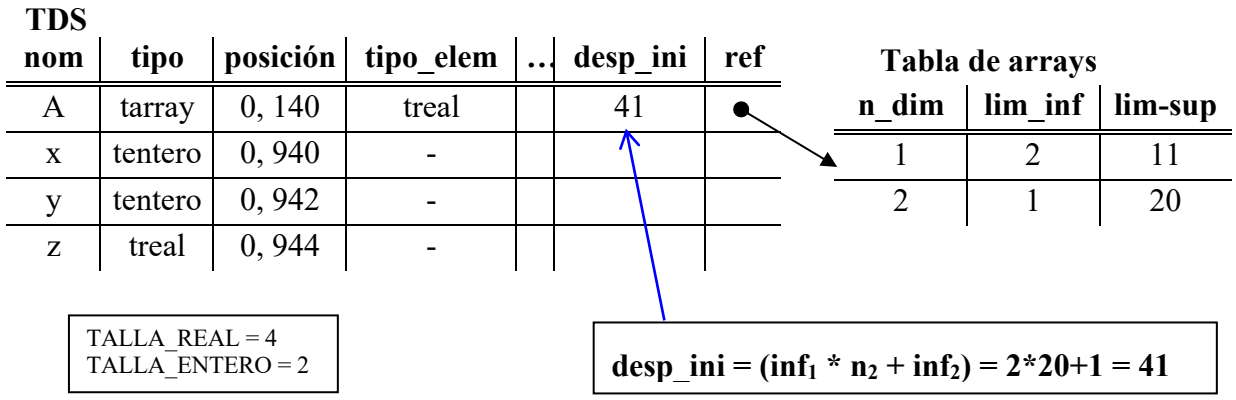


Figura 2.7. Contenido de la TDS para el Ejemplo 2.3.

Suponiendo que el contenido en la TDS es el de la Fig. 2.7 se generará el siguiente código intermedio:

```
(100)   t1 := x
(101)   t1 := t1 * 20
(102)   t1 := t1 + y
(103)   t1 := t1 - 41
(104)   t1 := t1 * 4
(105)   A [t1] := z
```

2.2.2. Acceso a los elementos de una matriz multidimensional en C

La generación de código intermedio para acceder a matrices multidimensionales en C es más sencilla que la del caso de Pascal, ya que en C todas las matrices comienzan en la posición 0. Se trata por lo tanto de un caso particular del visto para Pascal.

En general, dada la declaración de una matriz n-dimensional en C

`int A[n1][n2]... [nn];`

la fórmula para acceder al elemento `A[i1][i2]...[in]` quedaría como:

$$base + (((i_1 * n_2 + i_2) * n_3 + i_3 \ldots) * n_n + i_n) * Talla$$

donde n_i es el número de elementos de la i-ésima dimensión.

Se puede construir un ETDS que genere código para el acceso a los elementos de una matriz n-dimensional en Pascal tal y como aparece en la Fig. 2.8.

IS -> **id**	{ LI.nom := id.nom }
LI = E	{ emite (LI.pos := LI.pos '*' Talla (id.nom)) ; IS.pos := BuscaPos (id.nom) ; emite (IS.pos '[' LI.pos ']' := E.pos ; }
LI → **[** E **]**	{ LI.pos := CrearVarTemp() ; emite (LI.pos := E.pos); LI.ndim := 1; }
LI →	{ LI_1.nom := LI.nom ; }
LI_1 **[** E **]**	{ LI.ndim := LI_1.ndim + 1; LI.pos := LI_1.pos ; emite (LI.pos := LI.pos '*' Num_elementos(LI.nom, LI.ndim)) ; emite (LI.pos := LI.pos '+' E.pos) ; }

Figura 2.6. ETDS para el acceso a los elementos de una matriz en C.

3. Expresiones lógicas e instrucciones de control de flujo

En un lenguaje de alto nivel las expresiones de tipo lógico se emplean principalmente como expresiones condicionales en instrucciones de control de flujo. Debido a esto hay una dependencia elevada entre la forma de representar los valores de las expresiones lógicas y el código que se debe generar para traducir las instrucciones de control de flujo. Por esta razón vamos a ver en la misma sección la representación de los valores lógicos y la traducción de instrucciones de control de flujo.

3.1. Variables y expresiones lógicas

Cualquier variable necesita una posición de memoria en la que, durante la ejecución del código generado, se almacene su valor. Cuando se trata de una *variable de tipo lógico o boolena*, para representar su valor también se necesita una posición de memoria (¡bastaría un bit!) para almacenar su valor. Como el código intermedio definido en este capítulo no dispone del tipo lógico ni el tipo bit, en la posición de memoria de una variable lógica se almacenará un entero que representará su valor. Para esta representación numérica se empleará el mismo criterio seguido en el lenguaje C[1] de que un cero representa el valor de verdad "falso", mientras que cualquier valor distinto de cero representa el valor de verdad "cierto".

Pero además de variables lógicas, en los lenguajes de alto nivel aparecen otro tipo de expresiones lógicas, conteniendo comparaciones y/o conectivas lógicas. Para representar el valor de una *expresión lógica* se pueden emplear dos métodos: la representación numérica y la representación por control de flujo.

[1] El tipo bool (lógico) no aparece en el lenguaje C hasta el estándar C99 (a través de <stdbool.h>) y no todos los compiladores lo soportan. Por esta razón, las instrucciones de control de flujo en C utilizan expresiones enteras (0 es falso y distinto de cero es cierto) para decidir el flujo.

En la *representación numérica* de una expresión lógica toda expresión lógica tendrá asignada una posición de memoria donde se almacenará su valor de verdad: un entero igual a 0 para "falso" o distinto de cero para "cierto". Por el contrario, en la *representación por control de flujo* no se emplea una posición de memoria para representar el valor de verdad de la expresión. Su valor vendrá determinado implícitamente por la instrucción de código intermedio que se ejecute. Tal y como se afirma en [AHOA90], ninguno de los dos métodos es siempre mejor que el otro. Nos centraremos en la representación numérica.

3.2. Representación numérica

En la representación numérica a cada expresión lógica se le asigna una posición de memoria donde se almacena un valor numérico que indica el valor de verdad de la expresión. Si el código intermedio permite emplear valores y operadores binarios bastaría con usar un 0 para representar el valor falso y un 1 para el valor cierto. Si, como es nuestro caso, el código intermedio no permite el empleo de valores binarios, se puede usar un número entero con un valor de cero para representar el valor falso y un número distinto de cero para representar el valor cierto. Con esta forma de representación, la evaluación de una expresión lógica se convierte en la evaluación de una expresión aritmética.

En los siguientes ETDS se va a presentar una generación de código intermedio para el caso en el que el lenguaje fuente permite la utilización de expresiones booleanas (como en Pascal) en lugar de utilizar valores enteros (como en C). Como se ha comentado anteriormente, esto obligará a realizar una traducción entre la representación del lenguaje fuente (booleanos) y la del lenguaje intermedio (enteros).

Podemos escribir un ETDS para una sencilla gramática que genere expresiones lógicas similares a las de Pascal tal y como aparece en el Fig. 2.8.

E → **true**	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' 1) }
E → **false**	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' 0) }
E → **(** E_1 **)**	{ E.pos := E_1.pos }
E → **id**	{ E.pos := BuscaPos(id.nom) }

Figura 2.8. ETDS representación numérica de expresiones lógicas.

Tal y como se puede apreciar en el ETDS anterior, cuando una expresión lógica sea la constante TRUE o FALSE, basta con crear una nueva variable temporal y asignar en la posición de memoria correspondiente un 1 o un 0 respectivamente. En el caso de que la expresión lógica sea una variable (id), al igual que ocurría en el caso de las expresiones aritméticas, basta con que el atributo E.pos tome el valor de la posición de memoria donde se almacena el valor de verdad del identificador (devuelto por la función *BuscaPos*).

Para evaluar una expresión relacional, como `a<5` hay que tener en cuenta que su valor de verdad, en general, no se puede conocer en tiempo de compilación. Por esta razón es necesario generar código de tres direcciones cuya ejecución produzca como resultado la evaluación de la expresión relacional, asignando en la posición de memoria reservada para la expresión un valor 0 o distinto de cero según sea falsa o cierta la expresión, respectivamente.

Ejemplo

Para evaluar la expresión `a<5`, se puede generar la siguiente secuencia de código de tres direcciones:

```
(1) t1 := 1
(2) if a < 5 goto 4
(3) t1 := 0
(4) ...
```

Tal y como fácilmente se puede comprobar, la ejecución del fragmento anterior de código de tres direcciones produce como resultado que si `a < 5` se almacenará un 1 en la posición de memoria reservada para almacenar el valor de verdad de la expresión (en el ejemplo t1). En caso contrario se almacenará un 0. El ETDS para generar la secuencia de código intermedio anterior es

$E \rightarrow E_1$ **oprel** E_2	{ E.pos := CrearVarTemp() ; emite(E.pos '**:=**' 1) ; emite('if' E_1.pos oprel.op E_2.pos 'goto' SIGINST+ 2) ; emite(E.pos '**:=**' 0) }

En este ETDS puede verse la utilización del atributo oprel.op. Este atributo represente al valor del operador relacional que aparece en el código fuente que se está compilando: "=", ">", ">=", "<",… También se puede observar la aparición de una nueva variable global: *SIGINST*. En el código intermedio de este tipo de expresiones se debe generar una instrucción de salto condicional que se salte la instrucción que le sigue. Para generar esta instrucción se debe conocer cuál es el número de la instrucción que se está generando. Con este fin, en el ETDS (y por lo tanto en el código fuente del compilador desarrollado) existe un contador del número de instrucciones de código intermedio generadas: SIGINST. Esta variable global del compilador es actualizada de forma automática por el procedimiento *emite* cada vez que se genera una instrucción. De esta forma, si en el momento de llamar al procedimiento *emite* para generar la instrucción de salto condicional '**if**' E_1.pos **oprel.op** E_2.pos '**goto**' SIGINST+ 2, la variable SIGINST vale 15, se generará la instrucción `(15) if pos1 oprel pos2 goto 17.`

Cuando la expresión lógica incluye operadores lógicos "and" y "or", si el código intermedio no dispone de estos operadores, como es nuestro caso, será necesario buscar una equivalencia con los operadores disponibles en el código intermedio. En nuestro caso, podemos observar que el operador lógico "and" puede implementarse como una operación de multiplicación entre

valores de verdad, y el operador "or" como una suma. De esta manera, podemos obtener el ETDS:

$E \rightarrow E_1$ **or** E_2	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' E_1.pos '+' E_2.pos) }
| E_1 **and** E_2	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' E_1.pos '*' E_2.pos) }

Al generar código intermedio de esta manera hay que tener en cuenta que, cuando se realice una operación "or" entre dos expresiones con el valor de verdad "cierto", el resultado será otra expresión con el valor "cierto" pero representado con un valor entero distinto de "1". Esto introduce un nuevo problema: En algún momento podría darse el caso de que el valor entero sobrepase el rango válido de los enteros, apareciendo enteros negativos. Esto a su vez puede provocar que una operación "or" entre dos expresiones ciertas representadas por valores enteros iguales pero con signos distintos den como resultado el valor falso. Así por ejemplo, si consideramos dos expresiones booleanas con el valor "cierto" pero representadas respectivamente por los valores enteros "-5" y "5", la operación "or" entre ellas (su suma) dará como resultado "0" (valor "falso"). Esto es un error provocado por la forma en la que generamos el código y por lo tanto debe corregirse. Una posible forma de corregir este problema consiste en asegurarse de que un valor "cierto" siempre se representa por el entero "1". Para ello generaremos una instrucción que compruebe el valor entero después de la operación "or". De esta manera, el ETDS anterior quedaría:

$E \rightarrow E_1$ **or** E_2	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' E_1.pos '+' E_2.pos) emite('if' E.pos '<= 1' goto' SIGINST+2) ; emite(E.pos '**:= 1**') }

Para generar el código correspondiente al operador lógico "not", si no está disponible en el código intermedio, tal y como es nuestro caso, será necesario generar una secuencia de instrucciones cuya ejecución produzcan el resultado equivalente. Así por ejemplo, para obtener en la posición de memoria t1 el valor numérico de verdad de la expresión "not a" se puede generar la secuencia de instrucciones de código intermedio:

```
(1) t1 := 0
(2) if a <> 0 goto 4
(3) t1 := 1
(4) ...
```

El ETDS correspondiente quedará como aparece en la Fig. 2.9.

$E \rightarrow E_1$ **or** E_2	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' E_1.pos '+' E_2.pos) emite('if' E.pos '<= 1' goto' SIGINST+2) ; emite(E.pos '**:= 1**')}
| E_1 **and** E_2	{ E.pos:= CrearVarTemp() ; emite(E.pos '**:=**' E_1.pos '*' E_2.pos) }
| **not** E_1	{ E.pos:= CrearVarTemp() ; emite(E.pos ':=' 0) ; emite('if' E_1.pos '<> 0 goto' SIGINST+2) ; emite(E.pos ':=' 1) }

Figura 2.9. ETDS operadores lógicos con representación numérica.

Para ver un ejemplo de la utilización de las expresiones lógicas en las instrucciones de control de flujo, se va a construir un ETDS que genere código de tres direcciones para la instrucción de do-while de C. Su sintaxis es

$I \rightarrow$ **do** I_1 **while (** E **)**

El esquema del código a generar se puede representar tal y como aparece en la Fig. 2.10. El bloque I_1 representa todas las instrucciones que se han generado en las producciones del no-terminal I para el cuerpo del bucle. De igual forma, el bloque E representa todo el código intermedio emitido para la evaluación de la expresión E.

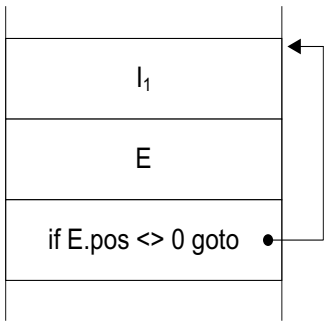


Figura 2.10: Esquema de código para un do-while con representación numérica.

La ejecución del código intermedio generado para la evaluación de la expresión lógica dejará, tal y como se ha visto, en una posición de memoria (E.pos) un valor 0 o distinto de 0 (1) según sea falsa o cierta la expresión. El código que se genera para la instrucción do-while, saltará al inicio del bucle si en esa posición hay un número distinto a 0. El ETDS quedará por lo tanto

I → **do**	{ I.inicio := SIGINST }
I_1	
while (E **)**	{ emite('if' E.pos '<> 0 goto' I.inicio) }

Se puede observar que se ha utilizado el atributo *I.inicio* para guardar el valor de la variable global SIGINST justo antes de I_1. Este atributo almacenará el número de la primera instrucción de código intermedio generada en I_1. Esta instrucción será el destino del salto condicional generado para ejecutar una nueva iteración del bucle. Este ETDS emite únicamente una instrucción, el salto que aparece al final del bucle, ya que las instrucciones del cuerpo del bucle serán emitidas por las acciones semánticas de las producciones I y las correspondientes a la evaluación de la expresión lógica por las del no-terminal E.

De forma semejante se puede intentar construir una gramática atribuida que genere código de tres direcciones para una instrucción "while" de C:

I→ **while**	{ I.inicio:= SIGINST }
(E **)**	{ emite('if' E.pos '=0 goto' I.fin) }
I_1	{ emite('goto' I.inicio) ;
	I.fin := SIGINST }

Sin embargo, se puede observar que esta gramática no es L-atribuida. Esto es debido a que en la acción semántica de generación de la instrucción de salto condicional { emite('if' E.pos'=0 goto' I.fin) } aparece el atributo I.fin que toma su valor en una acción que aparece a la derecha de la acción semántica del *emite*. Esto significa que si se realiza una traducción dirigida por la sintaxis, en el momento en que se realiza la llamada a *emite* para generar la instrucción de salto condicional, no se conoce todavía el valor del atributo I.fin que debería contener el número de la primera instrucción de código intermedio generada después del bucle (se trata de un argumento de la llamada a *emite* cuyo valor no se conoce). Este problema de tener que generar una instrucción antes de conocer todos sus argumentos se resolverá en la sección dedicada a las referencias no satisfechas.

4. Referencias no satisfechas. Relleno por retroceso

4.1. Planteamiento del problema

En la sección anterior se ha visto que frecuentemente se debe generar una instrucción de código de la que no se conoce el valor de todos sus argumentos. Una primera aproximación para solucionar este problema es realizar dos pasadas: En una primera pasada se generan las instrucciones aunque algunas de ellas contengan referencias cuyo valor no se conoce todavía. Tras esta pasada, en la que se han generado todas las instrucciones, se realiza una segunda pasada en la que se van sustituyendo las referencias por sus valores.

Una alternativa que permite solucionar el problema de las referencias no-satisfechas sin tener que realizar una segunda pasada es la conocida como *relleno por retroceso*. Básicamente esta técnica consiste en la generación de cada instrucción aunque alguno de sus argumentos quede incompleto. Cada vez que se genera una instrucción incompleta se guarda el número de la instrucción incompleta en un atributo relacionado con al argumento cuyo valor no se conoce. Cuando posteriormente, al seguir generando el resto de las instrucciones, se conozca el valor se incorpora directamente a la instrucción que quedó incompleta, quedando de esta manera completada.

4.2. Relleno por retroceso

Para realizar la implementación de esta técnica se pueden definir tres funciones con el siguiente perfil:

ptro CreaLans(E): Crea una lista que solo contiene un número de instrucción `E` a rellenar posteriormente. Devuelve un puntero a dicha lista.

void CompletaLans(ptro, E): Rellena todas las instrucciones incompletas, cuyo número está contenido en la lista apuntada por `ptro`, con el valor de argumento `E`.

ptro FusionaLans (ptro, ptro): Concatena las listas apuntadas por sus dos argumentos y devuelve un puntero a la nueva lista.

Veamos mediante un ejemplo cómo pueden utilizarse estas funciones para implementar la técnica de relleno por retroceso. Supongamos que en el momento de generar la instrucción de salto incondicional de la línea 100 no se conoce cuál será el número de la instrucción destino de este salto.

```
100 goto -
```

Se puede anotar en una lista, relacionada con el destino del salto, el número de la instrucción generada que contiene este argumento no-satisfecho. Esto se hace mediante una llamada a la función `CreaLans(100)`. La lista ahora contendrá el número de la instrucción 100: lista={100}

Se continúa generando código y, si se genera otra instrucción de salto incompleta (por ejemplo la número 103) cuyo destino sea el mismo, se añade a la lista asociada a ese destino el número de la nueva instrucción incompleta: lista={100, 103}.

```
101 ...
102 ...
103 if  a>0 goto ---
```

Continúa la generación de código intermedio y, en el momento en el que se conozca el valor del argumento (por ejemplo el destino de los dos saltos podría ser la instrucción número 105), se

llama a la función `CompletaLans(lista ,105)` que se encargará de completar las instrucciones 100 y 103 con el valor 105. El código del ejemplo quedará por tanto:

```
100 goto 105
101 ...
102 ...
103 if  a>0 goto 105
104 ...
105 ...
```

En la sección 3.1 se construyó un ETDS para la instrucción while de C suponiendo que se utilizaba una representación numérica para las expresiones lógicas:

I→ **while**	{ I.inicio:= SIGINST }
(E **)**	{ emite('if' E.pos '=0 goto' I.fin) }
I_1	{ emite('goto' I.inicio) ;
	I.fin := SIGINST }

En este ETDS ya aparece el problema de tener que generar una instrucción de salto condicional en la que no se conoce el número de la instrucción destino. Ahora se puede resolver este problema usando relleno por retroceso. Justo antes de generar la instrucción de salto se crea una lista en la que se guarda el número de la instrucción que contiene la referencia no-satisfecha: `CreaLans(SIGINST)`. Cuando finalmente se conozca el destino del salto (la instrucción que se genere justo después de la última instrucción del bucle), se llamará a la función `CompletaLans(final, SIGINST)` para completar la instrucción de salto condicional. El ETDS quedará por tanto como aparece en la Fig. 2.11.

I→ **while**	{ I.inicio:= SIGINST }
(E)	{ I.final:= CreaLans(SIGINST) ;
	emite('if' E.pos '=0 goto' ---) }
I_1	{ emite('goto' I.inicio) ;
	CompletaLans(I.final, SIGINST) }

Figura 2.11. ETDS *while* con relleno por retroceso (representación numérica de expresiones lógicas).

4.3 Representación numérica de expresiones lógicas (con relleno por retroceso)

Usando relleno por retroceso se puede resolver también el problema de argumentos no satisfechos que aparece al intentar generar código para otras instrucciones de control de flujo en C. Suponiendo que se emplea una representación numérica de las expresiones lógicas, se puede generar código intermedio para una instrucción if-else de C mediante el siguiente ETDS:

I → **if** E	{ I.falso:= CreaLans(SIGINST); emite('if' E.pos '=0 goto' ---); }
I_1 **else**	{ I.fin := CreaLans(SIGINST) ; emite('goto' **---**) ; CompletaLans(I.falso, SIGINST) }
I_2	{ CompletaLans (I.fin, SIGINST) }

En este ETDS, además del salto al código del "else" para el caso en que la expresión sea falsa, se emite un salto "emite('goto' **---**)" que garantiza que, tras la ejecución del código de I_1 no se ejecuta siempre el del I_2. Las dos instrucciones de salto se generan de forma incompleta (falta el destino de los saltos) y se completan posteriormente con las llamadas a "CompletaLans(I.falso, SIGINST)" para añadir el destino del salto a I_2 y con "CompletaLans (I.fin, SIGINST)" para completar el salto emitido después del código de I_1.

De forma similar, podemos construir un ETDS que genere código intermedio para un bucle for parecido al del lenguaje C:

I → **for (** I_1 **;**	{ I.cond := SIGINST; }
E **;**	{ I.fin := CreaLans (SIGINST); emite('if' E.pos '=0 goto' ---); I.cuerpo := CreaLans (SIGINST); emite('goto' **---**) ; I. incr := SIGINST ; }
I_2 **)**	{ emite('goto' I.cond); CompletaLans(I.cuerpo, SIGINST) ; }
I_3	{ emite('goto' I.incr); CompletaLans(I.fin, SIGINST) ; }

5. Subprogramas

En el capítulo anterior ya se vio con bastante detalle cuál sería la secuencia de carga y descarga de un registro de activación para la llamada y finalización de un subprograma. En esta sección se presentan los ETDS que especifican la generación del código intermedio que se encarga de realizar la carga y descarga de un registro de activación (Pascal) y marco (C).

5.1. Registros de activación en Pascal.

5.1.1. Declaración

Cuando se genera código para la declaración de un subprograma es necesario generar el código que realiza la parte de la secuencia de carga y descarga del registro de activación que lleva a cabo el subprograma llamado. Se muestra a continuación como quedaría esta parte del ETDS correspondiente a la declaración a una función en Pascal. Obsérvese como es necesario utilizar relleno por retroceso para completar la instrucción de salto incondicional encargada de saltar el posible código de subprogramas locales al subprograma que se está compilando.

La instrucción PUSH permite apilar el valor del display del nivel de anidamiento correspondiente al subprograma que estamos compilando (PUSH Display[NIVEL]). Así mismo, suponemos que el tope de la pila se encuentra almacenado en la posición (o registro) TOP.

DS → **function id (**Param_Form**):**
Tipo**;**
 	
Bloque **;**
 	
Decl_Subprg
	
	{emite('push Display['NIVEL']');
	emite('display['NIVEL'] := top');
	DS.area_datos := CreaLans(SIGINST);
	emite('incTop' ---) ; }
	{ CompletaLans (DS.area_datos, DESP)
	emite('top := Display['NIVEL']');
	emite('pop Display['NIVEL']');
	emite('ret') }

Bloque → Decl_Var Decl_Subprg **begin** Instrucciones **end**	{ Bloque.inst := CreaLans(SIGINST); emite('goto' ----); } { CompletaLans(Bloque.inst,SIGINST); }

Figura 2.18. ETDS carga/descarga de un registro de activación por el procedimiento llamado.

5.1.2. Llamada

Cuando se genera código para la llamada a un subprograma es necesario generar el código que realiza la parte de la secuencia de carga y descarga de un registro de activación que lleva a cabo el subprograma llamador. Se muestra a continuación como quedaría el ETDS correspondiente al lenguaje Pascal. Obsérvese como se utiliza la instrucción INCTOP para reservar espacio para el valor devuelto por la función, y la instrucción PUSH para apilar los parámetros actuales. Posteriormente, durante la secuencia de descarga del registro de activación, se utiliza una instrucción DECTOP para decrementar el valor del tope de la pila y de esta forma desapilar los parámetros que se cargaron al realizar la llamada.

E → **id** **(** Param_Act **)**	{ E.pos := CreaVarTemp(); emite('incTop Talla_Tipo(id.nom)) } { emite('push' <estado máquina>); emite('call' BuscaPos(id.nom)) ; emite('pop' <estado máquina>); emite('decTop' TallaParam(id.nom)) ; emite('pop' E.pos) }
Param_Act → E	{ emite('push' E.pos) }
Param_Act → Param_Act **,** E	{ emite('push' E.pos) }

Figura 2.19. ETDS carga/descarga de un registro de activación por el procedimiento llamador.

5.2. Marcos en C

5.2.1. Declaración

Se muestra a continuación como quedaría un ETDS correspondiente a la declaración a una función en C. La instrucción "push FP" que se emite, permitirá en tiempo de ejecución apilar el valor del "frame pointer" correspondiente al subprograma que se estará cargando.

DS → Tipo **id (** Param_Form **)**
 	
Bloque **;**
	{ emite('push FP');
	emite('FP := TOP');
	DS.area_datos := CreaLans(SIGINST);
	emite('TOP := TOP + ---') ; }
	{ CompletaLans (DS.area_datos, DESP)
	emite('TOP := FP');
	emite('FP := pop);
	emite('ret') }

Figura 2.18. ETDS carga/descarga de un frame en C realizado por la función llamada.

5.2.2. Llamada

Al igual que en Pascal, en C cuando se genera código para la llamada a una función es necesario generar el código que realiza la parte de la secuencia de carga y descarga del frame de la función llamada. La instrucción "top := top + Talla_Tipo(id.nom)" se emite para que al ejecutarse reserve espacio para el valor devuelto por la función. De igual forma, la instrucción "push" se emite para que en tiempo de ejecución apile los parámetros actuales. Durante la secuencia de descarga del frame se utiliza decrementa el valor del tope de la pila para, de esta forma, desapilar los parámetros que se cargaron al realizar la llamada.

E → **id** **(** Param_Act **)**	{ E.pos := CreaVarTemp(); emite('TOP := TOP +' Talla_Tipo(id.nom)) } { emite('push' <estado máquina>); emite('call' BuscaPos(id.nom)) ; emite('pop' <estado máquina>); emite('TOP := TOP -' TallaParam(id.nom)) ; emite('pop' E.pos) }
Param_Act → E	{ emite('push' E.pos) }
Param_Act → E , Param_Act	{ emite('push' E.pos) }

Figura 2.19. ETDS carga/descarga de un frame realizado por la función llamadora.

Bibliografía recomendada

El tema de generación de código intermedio que más coincide con el contenido de este tema es el que aparece en [Aho 2008].

[Aho 2008] Aho, Alfred V.; Lam, Monica S.; Sethi, Ravi; Ullman, Jeffrey D. Compiladores: Principios, técnicas y herramientas. Segunda edición Pearson/Addison-Wesley Iberoamericana, 2008

Ejercicios

A continuación se muestra una serie de ejercicios recopilados en su mayoría de exámenes. Las soluciones propuestas (ejercicios marcados con un asterisco) se han dejado a modo de ejemplo. Es posible que alguna de las soluciones mostradas no coincida con el nivel o la forma en la que se presenta actualmente la materia.

***9.1.-** La siguiente gramática genera bucles "for" muy parecidos a los de Pascal estándar pero con algunos matices:

I → **for id :=** E **to** E S **do** I
  | **begin** LI **end**

S → **step** E
  | ε

LI → I **;** LI
  | I

...

donde E es una expresión de tipo entero.

La opción "step E" que puede aparecer en la cabecera del bucle indica que la variable contador (id) en vez de incrementarse en una unidad (como ocurre cuando no aparece la opción ("step"), se incrementará en "E" unidades al final de cada iteración.

Construir un ETDS que incluya las acciones semánticas necesarias para la comprobación de tipos y la generación de código intermedio. Deben emplearse listas de referencias no satisfechas (relleno por retroceso) para el control de los saltos.

***9.2.-** Dado el siguiente fragmento de una gramática que genera instrucciones de asignación con sumas y multiplicaciones, donde las llaves "{ }" representan la clausura reflexivo-transitiva (0 o más ocurrencias):

IS → **id :=** E

E → T { **+** T }

T → F { * F }

F → **num**
  | **id**
  | **(** E **)**

...

a) Construir un ETDS para la generación de código intermedio.

b) Partiendo del ETDS anterior, incluir en el analizador descendente recursivo asociado a la gramática, las acciones necesarias para la generación de código intermedio.

NOTAS:

- No incluir comprobaciones de tipo.
- Emplear el juego de instrucciones de tres direcciones. (Ejemplo: "id.pos:=id.pos*5")
- El analizador léxico devuelve para cada identificador su entrada en la TDS en la variable "ind", y para una constante numérica su valor en la variable global "num".
- Se suponen definidas las siguientes funciones:
- *CrearVarTemp()*: Crea una variable temporal y devuelve su posición.
- *BuscaPos(id.nom)*: Devuelve la posición de memoria asignada al identificador cuya entrada en la TDS es id.nom.
- En la declaración de cada parámetro formal, deberá indicarse si el uso es por valor o referencia.

***9.3.-** Dada la siguiente gramática (inspirada en la proposición "switch" del C):

I → **switch (** E **) {** L **}**
L → **case num :** P **;** L
L → **default :** P **;**
L → **ε**
...
P → P **;** P
P → **break**
P → I
P → **ε**
...

Donde:

a) Tanto "**E**" como la constante "**num**" deben ser enteras.

b) Si la constante de un "case" coincide con el valor de la expresión E, se deberá comenzar a ejecutar todas las instrucciones asociadas a ese "case" y a los que le siguen. La proposición "break" provoca una salida inmediata del "switch". Las instrucciones asociadas al "default" se ejecutarán siempre (excepto si se encontró antes una instrucción "break").

Construir un ETDS que realice las acciones semánticas necesarias para la generación de código intermedio de tres direcciones.

***9.4.-** Construir un Esquema de Traducción Dirigido por la Sintaxis que realice las comprobaciones semánticas y genere código de tres direcciones para una gramática con las siguientes reglas de producción.

S → **case** E **of** L **end**

L → **num :** S **;** L

L → **ε**

...

Tanto las constantes de selección, como la expresión, deben ser de tipo entero.

***9.5.-** Dada la siguiente gramática:

P → D **;** I **.**

D → **id :** T | D **;** D | ε

T → **pila (** num **) de** TS | TS

TS → **entero** | **real**

I → **apilar (** **id ,** E **)** | **id :=** E | I **;** I | ε

E → **desapilar (id)** | **cima (id)** | **id**

Donde:

pila (num) de TS, representa la definición del tipo pila, con un máximo de **num** elementos de tipo entero o real.

apilar (id, E), indica que se apila el valor de E en la pila **id**

desapilar (id), devuelve el elemento que desapila de la pila **id**

cima (id) devuelve el elemento que hay en la cima de la pila **id** (no lo desapila).

Construir un ETDS que realice las acciones semánticas necesarias para la comprobación de tipos, la asignación de memoria y la generación de código tres direcciones teniendo en cuenta que:

a) La talla de los enteros y reales es 1.

b) No se debe permitir la asignación de pilas.

c) No hay coerción de entero a real.

Se deben realizar las comprobaciones dinámicas para los controles de desborde de pila y pila vacía.

***9.6.-** Dada la siguiente gramática:

P → D **;** I **.**
D → **id :** T | D **;** D | **ε**
T → **cadena (num)** | TS
TS → **entero** | **real** | **caracter**
…
I → **concatena (id , id)** | **id :=** E | I **;** I | **ε**
E → **longitud (id)** | **id**

Donde:

cadena(num) representa la definición de un objeto de tipo "cadena de caracteres" cuyo tamaño máximo viene dado por **num**.

concatena(id1, id2) concatena al final de la cadena id1 la cadena id2.

longitud(id) devuelve el tamaño de la cadena id.

Construir un ETDS que realice las acciones semánticas necesarias para la comprobación de tipos, la asignación de memoria y la generación de código de tres direcciones teniendo en cuenta que:

a) La talla de los enteros, reales y caracteres es 1, y no hay coerción entre tipos.

b) Se deben realizar las comprobaciones dinámicas para controlar que no se sobrepasa el tamaño máximo permitido de una cadena.

c) No se deben permitir las asignaciones entre cadenas.

NOTA: Se recomienda implementar una cadena declarada de tamaño máximo num, como un vector de num+1 elementos, donde el elemento 0 se puede usar para almacenar la longitud actual de la cadena.

***9.7.-** Incorporar a la gramática siguiente, las acciones semánticas necesarias para la comprobación de tipos, la asignación de memoria y la generación de código intermedio (tres direcciones):

S → D S

D → D **;** D | **id :** T

T → **char** | **integer** | **string(num)**

S → S **;** S | **id :=** E

E → F **+** F | **id (num : num)**

F → **id** | **/id/**

NOTAS:

1) string(num) representa una cadena de caracteres de talla máxima num.

2) id(a:b) representa la subcadena de id comprendida entre a y b.

3) /id/ es la longitud de la cadena id.

4) Será necesario considerar explícitamente la talla actual de las cadenas.

***9.8.-** Dado el siguiente fragmento de gramática.

I → **while (** E **)** I Rw

Rw → **except** E **do** I

Construye un ETDS para la comprobación de tipos y emisión de código intermedio. La sentencia "while" tiene el comportamiento estándar de Pascal, pero con un pequeño matiz: La parte "except" permite indicar una instrucción alternativa a ejecutar en las iteraciones en las que se cumple la condición booleana introducida por la palabra "except". La expresión del "except" se evalúa antes de entrar en el cuerpo del bucle.

Ejemplo:

```
a = 9 ;
while ( a < 15 )
  { a= a+1;  write(a) } => 10 11 12 13 15
except a==13 do a= a+1;
```

9.9.- Escribe un ETDS que genere código intermedio para el siguiente fragmento de una gramática independiente del contexto, correspondiente a un lenguaje de programación. La semántica de la instrucción **loop** es la siguiente: Si la expresión "E" que sigue a **loop** es cierta se ejecutan las instrucciones que siguen al **then** y preceden a **exitloop**, y la instrucción **loop** finaliza, en caso contrario, se irán evaluando sucesivamente las expresiones que siguen al **elseif** hasta que una sea cierta, en cuyo caso, se ejecutan las instrucciones que siguen a ese **then** y el flujo del programa vuelve al principio de la instrucción **loop**. Si ninguna expresión es cierta se ejecutan las instrucciones que siguen al **else** y se vuelve al principio de la instrucción **loop**.

I → **loop** E **then** I **exitloop** O

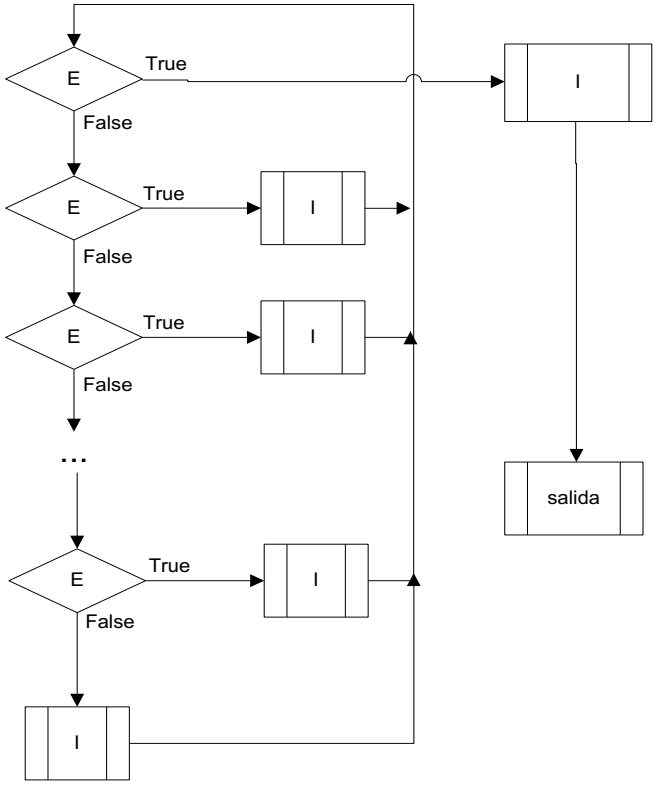
O → **elseif** E **then** I O

O → **else** I **endloop**

I → I **;** I

E → E oprel E

El diagrama de flujo de la instrucción sería el de la figura.



***9.10.-** Definir un ETDS que realice la generación de código intermedio para la instrucción WHILE, en la que se permita la posibilidad de una instrucción EXITLOOP. La instrucción EXITLOOP supone la salida inmediata del bucle. Utiliza la sintaxis propia de C.

***9.11.-** Construir un ETDS que realice la declaración, la comprobación estática de tipos y la generación de código intermedio para la siguiente gramática:

P → D ; I .

D → id : T | D ; D | ϵ

T → integer | set of [num]

E → member (E , id) | num | id

...

I → for-each id in id do I

Siendo:

set of [num] un tipo de datos que representa un conjunto cuyos elementos pueden ser {0, 1, 2, 3, . . . num}. Por ejemplo: "a: set of [5]" declara la variable a cuyo valor puede ser cualquier subconjunto de {0, 1, 2, 3, 4, 5}.

member(E, id) una función que devuelve TRUE (ó 1) si el elemento E está en el conjunto id (FALSE ó 0, en caso contrario).

for-each id_1 in id_2 do I una instrucción donde la variable entera "id_1" toma todos los valores que corresponden a la variable conjunto id_2 . Además, la instrucción I se ejecutará para cada uno de los valores de id1 cuando este valor pertenezca al conjunto id_2. Por ejemplo:

```
a: set of [5];
.....
a := {1, 3, 5}
for-each n in a do print (n);
```

producirá como resultado: 1, 3, 5.

Nota.- Se recomienda representar las variables de tipo conjunto "set of [num]" como un "num + 1" posiciones consecutivas de memoria. Donde cada posición de memoria representará la pertenencia (1 ó 0) de un elemento al conjunto. Por ejemplo, la variable "a", definida anteriormente, se representaría por 6 posiciones de consecutivas memoria, que si contuviesen los valores (1, 0, 1, 0, 0, 1) indicarían que el conjunto a contiene los elementos {0, 2, 5}.

9.12.- Dada la siguiente gramática, construir un ETDS que realice: las comprobaciones semánticas y de tipo usuales en la instrucción FOR y la generación de código intermedio.

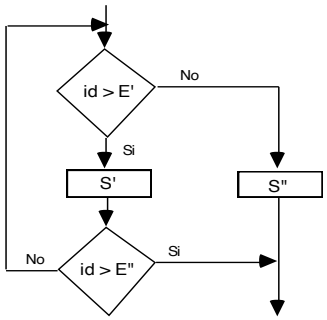
S → for id := E R E do S

R → to | downto

9.13.- Construir un ETDS que genere código intermedio para comprobar en tiempo de ejecución que los índices de una matriz están dentro de los límites definidos en su declaración. Para la generación de este código, se puede suponer la existencia de otros atributos necesarios, cuya evaluación se realiza en otras producciones, siempre que se indique su carácter sintetizado o heredado.

9.14.- Considérese la siguiente instrucción cuya semántica viene dada por el diagrama de flujo que se adjunta. Diseñar el ETDS para la generación de código tres direcciones.

S → **pseudowhile id ,** E' **until** E" **do** S' **then** S"



9.15.- Dada la siguiente gramática que representa registros en Pascal, construir un ETDS que realice:

P → D **;** I
D → D **;** D | **id :** T
T → **integer** | **real** | **record** LC **end**
LC → **id :** T RLC
RLC → **;** LC | ε
I → L **:=** E
L → L **. id** | **id**
E → E + E | L

a) Las comprobaciones semánticas de: Compatibilidad de tipos, los tipos de los campos de un RECORD solo pueden ser de tipo simple y no se permite la asignación de registros.

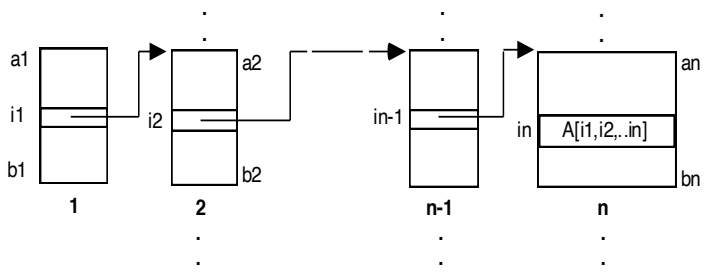
b) El cálculo de las posiciones relativas de los objetos en memoria.

c) La generación de código intermedio para las expresiones e instrucciones.

d) Extender el ETDS para generar código intermedio para la asignación de registros.

e) Extender el ETDS para permitir registros con parte variante.

9.16.- Suponiendo que una matriz A[a1..b1,a2..b2,...an..bn] se representa en memoria por el método de división en subtablas (tal y como se ilustra en la figura). En las subtablas de los niveles 1..n-1 únicamente aparecen referencias (de talla 1) a los orígenes de las subtablas de los niveles inmediatamente siguientes. Solo en las subtablas de nivel n se almacenarán los elementos de la matriz.



a) Determinar la función de acceso para el elemento genérico A[i1,i2,...in].

b) Diseñar el ETDS para la generación del código intermedio correspondiente, para la siguiente gramática.

S → L **:=** E
E → L
L → **id**
L → I **]**
I → I **,** E
I → **id** **[** E

9.17.- Dada la siguiente gramática:

P→ **var** D_V I

D_V → D_V D_V | Lista_id **:** T **;**

Lista_id → Lista_id **, id** | **id**

T → **record** ListaCampos Cvar **end** | T_simple

T_simple → T_disc | **integer** | **real**

ListaCampos → ListaCampos **;** ListaCampos | **id :** T_simple

T_disc → **num..num** | **boolean**

Cvar → **case id :** T_disc **of** LCV | ε

LCV → LCV **;** LCV | Val_sel **: (** ListaCampos **)**

Val_sel → **num** | **id**

I → **id :=** E | **id . id :=** E | **if** E **then** I **else** I

E → E **oprel** E | **id . id** | **id**

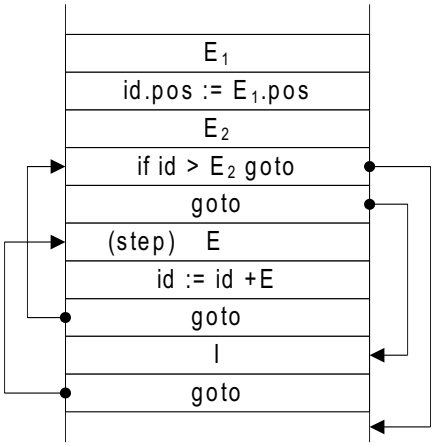
a) Dar un ETDS que realice el cálculo de las posiciones relativas de las variables en memoria para esta gramática.

b) Ampliar el anterior ETDS para que realice la comprobación de tipos y genere el código intermedio necesario para evaluar las expresiones y realizar las asignaciones adecuadas en dicha gramática.

NOTA: Diseñar la estructura de la tabla de símbolos que permita la realización de los apartados anteriores. Igualmente, describir brevemente los atributos, así como los perfiles y acciones realizadas por las funciones y/o procedimientos que utilicéis.

Soluciones a los ejercicios seleccionados

9.1.-



I → **for id :=** E_1	{ si E_1.tipo <> tentero ent MemError() id.pos:=BuscaPos(id.nom); emite(id.pos '**:=**' E_1.pos) }
to E_2	{ si E_2.tipo <> tentero ent MemError() I. fin:=CreaLans(SIGINST); I.comi:=SIGINST; emite('if' id.pos '>' E_2.pos 'goto' ---) I.cont:=CreaLans(SIGINST); emite('goto' --); I.inc:=SIGINST; S.pos:=id.pos; }
S	{ emite('goto' I.comi); CompletaLans(I.cont, SIGINST) }
do I	{ emite('goto' I.inc); CompletaLans(I.fin, SIGINST) }
S → **step** E	{ si E.tipo <> tentero ent MemError() emite(S.pos '**:=**' S.pos '+' E.pos) }
S → ε	{ emite(S.pos '**:=**' S.pos '+' 1) }

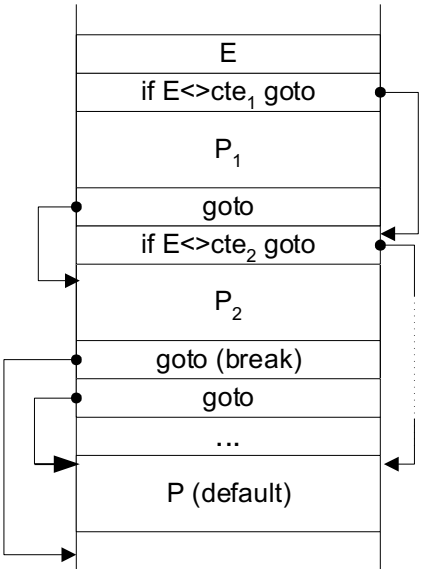
9.2.-

a)

E → T_1	{ E.pos:=CreaVarTemp(); emite (E.pos '**:=**' T_1.pos) }
{ +	
T_2	{ emite(E.pos '**:=**' E.pos '+' T_2.pos) }
}	
T → F_1	{ T.pos:=CreaVarTemp(); emite (T.pos '**:=**' F_1.pos) }
{ *	
F_2	{ emite(T.pos '**:=**' T.pos '*' F_2.pos) }

}	
F → **num**	{ F.pos:=CrearVarTemp(); emite(F.pos '**:=**' num.val) }
F → **id**	{ F.pos:=BuscaPos(id.nom) }
F → **(** E **)**	{ F.pos:=E.pos }
IS → id **:=** E	{ id.pos:=BuscaPos(id.nom); emite(id.pos '**:=**' E.pos) }

9.3.-

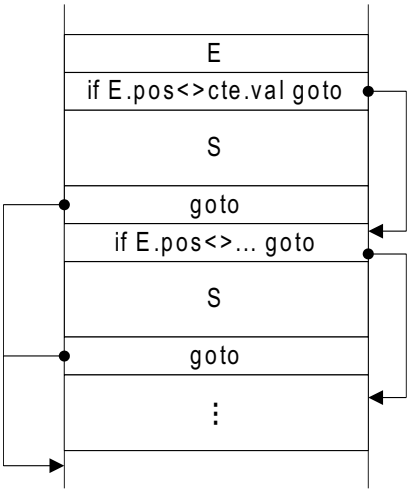


En la solución propuesta se puede observar que se genera una instrucción "goto" inmediatamente a continuación del cuerpo de un "case" cuya finalidad es saltar al cuerpo del siguiente "case". Con esto se consigue que si se cumple la igualdad de la constante de un "case", tras ejecutarse su cuerpo se ejecuten las instrucciones del cuerpo de todos los "case" que le siguen (hasta que se encuentre un salto generado para un break).

I → **switch (** E **)** { L }	{ L.pos:=E.pos; L.anterior := nil; L.fin:=nil}
L → **case num** : P ; L_1	{ I.sigcond:=CreaLans(SIGINST) ; emite('if' num.val '<>' L.pos 'goto' --); CompletaLans(L.anterior, SIGINST) } { L_1.anterior:=CreaLans(SIGINST) ; emite ('goto' --); L_1.pos:=L.pos; CompletaLans(I.sigcond, SIGINST); L_1.fin:=FusionaLans(P.fin, L.fin) }
L → **default** :	{ CompletaLans(L.anterior, SIGINST) }

P ;	{ CompletaLans(L.fin, SIGINST); CompletaLans(P.fin, SIGINST) }
L → **ε**	{ CompletaLans(L.anterior, SIGINST) CompletaLans(L.fin, SIGINST);}
P → P_1 ; P_2	{ P.fin:=FusionaLans(P_1.fin,P_2.fin) }
P → **break**	{ P.fin:=CreaLans(SIGINST); emite('goto' --) }
P → I	{ P.fin:=I.fin }
P → **ε**	{ P.fin:=nil }

9.4.-



L → **num** :	{ si num.tipo ≠ tentero ent MemError() ; I.sig:=CreaLans(SIGINST); emite('if' num.val '<>' L.pos 'goto' --) }
S ;	{ I.falsa:=CreaLans(SIGINST); emite('goto' --); CompletaLans(I.sig, SIGINST); L_1.pos:=L.pos }
L_1	{ L.fin:=FusionaLans(I.falsa, L_1.fin) }
L → **ε**	{ L.fin:=nil }
S → **case** E	{ si E.tipo ≠ tentero ent MemError(); L.pos:=E.pos }
of L	{ CompletaLans(L.fin, SIGINST) }

9.5.-

Es importante observar que en tiempo de compilación no conocemos el tamaño de la pila, solo conocemos el tamaño máximo de la pila. Por lo tanto, será necesario reservar una posición de memoria para que en tiempo de ejecución se almacene allí el tope de la pila. Podemos implementar la pila de **'num'** elementos como un vector (array) de **'num+1'**

elementos, donde el elemento 0 se puede usar para almacenar el tamaño actual (tope) de la pila. Cada vez que apilemos un nuevo elemento, lo haremos sobre la posición indicada en pila[0] (dirección base del vector). A continuación incrementaremos el valor de pila[0]. De esta forma, las acciones semánticas para generar código para apilar, desapilar, y obtener la cima podrían ser:

I → **apilar (id ,** E **)**	{ I.pos := BuscaPos (id.nom) ; emite (pos '**:=**' pos '+' 1); emite ('if' pos '<=' limite 'goto' SIGINST +2) emite ('fin') } { emite (I.pos '[' I.pos '] :=' E.pos) }
E → **desapilar (id)**	{ id.pos := BuscaPos (id.nom) ; emite ('if' id.pos '>' 0 'goto' SIGINST + 2) ; emite ('fin') ; E.pos := CrearVarTemp() ; emite (E.pos ':=' id.pos '[' id.pos ']'); emite (id.pos':=' id.pos '**-**' 1) }
E → **cima (id)**	{ id.pos := BuscaPos (id.nom) ; emite ('if' id.pos '> 0 goto' SIGINST +2); emite ('fin') E.pos := CrearVarTemp(); emite (E.pos ':=' id.pos '[' id.pos ']') }

En el ETDS anterior hemos se puede ver que `pos[0] y pos` hacen referencia al mismo valor, ya que es lo mismo tomar el contenido de la dirección base del vector (`pos`) o el contenido de la dirección resultante de sumar a la base del vector un desplazamiento cero (`pos[0]`).

El ETDS, incluyendo algunas comprobaciones de tipo quedará finalmente:

P → D **;** I **.**	{ DESP:=0 }
D → **id :** T	{ InsertarTDS (id.nom, T.tipo, DESP); DESP := DESP + T.talla ; si T.tipo = Tpila(limite,tipo) ent emite (DESP ':=' 0) }
D → D **;** D	
D → ε	
T → **pila (num) de** TS	{ si num.val <=0 ent T.tipo := terror sino T.tipo:= tpila (num.val, TS.tipo) ; T.talla := TS.Talla * (num.val +1) }
T → TS	{ T.talla := TS.talla ; T.tipo := TS.tipo }
TS → **entero**	{ T.talla := 1 ; T.tipo := tentero }

TS → **real**	{ T.talla := 1 ; T.tipo := treal }
I → **apilar (id ,** E **)**	{ si BuscaTipo (id.nom) <> tpila (limite,tipo) ent I.tipo := terror sino id.pos := BuscaPos (id.nom) ; emite (id.pos ':=' id.pos '+' 1); emite ('if' id.pos '<=' limite 'goto' SIGINST + 2) emite ('fin') si E.tipo <> tipo ent I.tipo := terror sino emite (id.pos'[' id.pos '] :=' E.pos) I.tipo := tvacío }
I → **id :=** E	{ id.tipo := BuscaTipo (id.nom) ; si id.tipo = tpila (limite, tipo) ent I.tipo := terror sino id.pos = BuscaPos (id.nom) ; si id.tipo <> E.tipo ent I.tipo := terror sino emite (id.pos ':=' E.pos) ; I.tipo := tvacio }
I → I **;** I	
I → ε	
E → **desapilar (id)**	{ si BuscaTipo (id.nom) <> tpila (limite, tipo) ent E.tipo := terror sino id.pos := BuscaPos (id.nom) ; emite ('if' pos '> 0 goto' SIGINST + 2) ; emite ('fin') ; E.pos := CrearVarTemp() ; emite (E.pos ':=' id.pos '[' id.pos ']'); emite (id.pos ':=' id.pos '-' 1) ; E.tipo := tipo }
E → **cima (id)**	{ si BuscaTipo (id.nom) <> tpila (limite, tipo) ent E.tipo := terror sino id.pos := BuscaPos (id.nom) ; E.pos := CrearVarTemp(); emite (E.pos ':=' id.pos '[' id.pos ']'); E.tipo := tipo }
E → **id**	{ E.tipo := BuscaTipo (id.nom) ; E.pos := BuscaPos (id.nom) }

9.6.-

P → { DESP:=0 } D **;** I **.**	
D → **id :** T | D **;** D | **ε**	{ InsertaTDS(id.nom, T.tipo, DESP) ; DESP:=DESP+T.talla }
T → **cadena (num)** | TS	{ T.tipo:=tcadena(num.val) ; T.talla:=num.val } { T.tipo:=TS.tipo ; T.talla:=TS.talla }
TS → **entero** | **real** | **caracter**	{ TS.tipo:=tentero ; TS.talla:=1 } { TS.tipo:=treal ; TS.talla:=1 } { TS.tipo:=char ; TS.talla:=1 }

I → **concatena (id_1 , id_2)**	{ id_1.pos:=BuscaPos(id_1.nom); id_2.pos:=BuscaPos(id_2.nom);
	si BuscaTipo(id_1.nom)=BuscaTipo(id_2.nom)=tcadena
	ent
	I.tipo:=tvacío;
	t1:=CrearVarTemp();
	emite (t1 ':=' 1);
	I.inicio:=SIGINST;
	I.listaFin:=CreaLans(SIGINST);
	emite ('if' t1 '>' id_2.pos 'goto' --);
	emite (id_1.pos ':=' id_1.pos '+' 1);
	emite ('if' id_1.pos'<='Talla(id_1.nom) 'goto' SIGINST+2);
	emite ('fin');
	emite (id_1.pos '[' id_1.pos ']:=' id_2.pos '[' t1 ']');
	emite (t1 ':=' t1 '+' 1);
	emite ('goto' I.inicio);
	CompletaLans (I.listaFin, SIGINST);
	sino I.tipo:=terror; }

| **id :=** E	{ si (BuscaTipo(id.nom):=E.tipo) ≠ tcadena ≠terror ent emite(BuscaPos(id.nom) ':=' E.pos); I.tipo:=tvacío; sino I.tipo:=terror; }
| I **;** I | **ε**	
E → **longitud (id)**	{ si BuscaTipo(id.nom) ≠ tcadena ent E.tipo:=terror; sino E.tipo:=tentero; E.pos:=BuscaPos(id.nom); }
| **id**	{ E.tipo:=BuscaTipo(id.nom); E.pos:=BuscaPos(id.nom); }

Se puede observar que en lugar de usar la técnica de relleno por retroceso con:

listaFin:=CreaLans(SIGINST);

emite ('if' t1 '>' id_2.pos 'goto' --);

en este caso podíamos haber contado las instrucciones que separaban la instrucción de salto y su destino, y haber emitido directamente la instrucción

*emite ('if' t1 '>' id_2.pos 'goto' SIGINST +7) */*

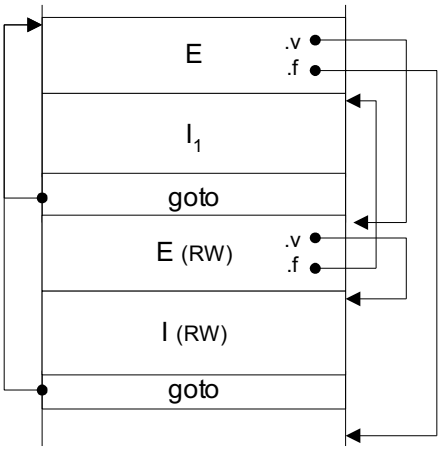
9.7.-

Las comprobaciones de tipo serán las típicas del bloque de declaraciones. Para asignar memoria basta con emplear una variable global que se incremente (según la talla del elemento definido) cada vez que se declare una variable. En el caso de una cadena de caracteres (tcadena), la talla nos viene determinada por el valor de la constante *num* (más uno por la razón que se explica más adelante).

Respecto a la generación de código intermedio, se trata de un ejercicio muy parecido a los dos anteriores. De forma parecida a como ocurría en ellos, en tiempo de compilación solo conocemos el tamaño máximo de la cadena, pero no su tamaño real en un instante dado de la ejecución. Una solución consistiría en reservar la primera posición de la cadena para que en tiempo de ejecución se almacene su tamaño. En este caso, cada vez que realicemos una asignación de cadenas, será necesario calcular el tamaño de la cadena resultado. Con este fin, se puede observar que la instrucción para obtener una subcadena: *id(num_1:num_2)* nos devuelve una cadena de tamaño num_2.*val* - num_1.*val* + *1*.

9.8.-

El esquema del código que queremos generar será:



El ETDS pedido quedaría:

I → **while** **(** E **)** I_1 RW	{ RW.inicio := SIGINST; } { si E.tipo <> tlógico ent MemError() ; I.fin = CreaLans(SIGINST); emite('if' E.pos '=0 goto' -) ; I.cond = CreaLans(SINGINST); emite('goto' -); RW.cuerpo := SIGINST ; } { emite ('goto' RW.inicio) ; CompletaLans(I.cond, SIGINST);} { CompletaLans (I.fin, SIGINST) ; }
RW → **except** E **do** I	{ si E.tipo <> tlógico ent MemError() ; emite('if' E.pos '= 0 goto' RW.cuerpo); } { emite ('goto' RW.inicio) ; }

9.10.-

Vamos a usar la gramática que se muestra a continuación, donde I representa una instrucción, y E una expresión. El no-terminal I tendrá más producciones, pero las que nos interesan son las relacionadas directamente con la generación de código intermedio para las instrucciones 'while' y 'exitloop'.

I → **while (** E **)** I

I → **exitloop**

I → I **;** I

I → …

Usaremos listas de referencias no satisfechas (relleno por retroceso) para el control de los saltos. El no-terminal E, que representa la expresión lógica, tendrá asociados dos atributos: E.lv y E.lf que representan respectivamente las listas de referencias no satisfechas de saltos para cuando la expresión es cierta (E.lv) y para cuando es falsa (E.lf).

SIGINST, es una variable global que contiene la etiqueta de la siguiente instrucción de código intermedio que se va a generar.

I → **while** **(** E **)** I_1	{ I.inicio := SIGINST } { CompletaLans (E.lv, SIGINST) } { emite ('goto' I.inicio) CompletaLans (I_1.salida, SIGINST) ; CompletaLans (E.lf, SIGINST) ; I.salida := nil }
I → **exitloop**	{ I.salida := CreaLans (SIGINST) ; emite ('goto' ---) }

$I \rightarrow I_1$ **;** I_2	{ I.salida:= FusionaLans (I_1.salida, I_2.salida) ; }

9.11.-

P → D **;** I .	
D → **id :** T	{ insertaTDS(id.nom, T.tipo) }
| D **;** D	
| **ε**	
T → **set of [num]**	{ T.tipo := conjunto(num.val) }
| **integer**	{ T.tipo := Tentero }
E → **member (** E_1 **, id)**	{ si (E1.tipo <> Tentero) or (BuscaTipo(id.nom) <> conjunto(n)) ent Terror; E.tipo := Tlógico; E.pos := CreaVarTemp(); emite(E.pos ':=' id.pos '[' E1.pos']') }
| **num**	{ E.tipo := Tentero; E.pos:= CreaVarTemp(); emite(E.pos ':=' num.val) }
| **id**	{ E.tipo := BuscaTipo(id.nom) E.pos := BuscaPos(id.nom) }

I→ **for_each** id_1 **in** id_2
 	
do I
	{ si (BuscaTipo(id_1.nom)<> Tentero)
	or (BuscaTipo(id_2.nom) <> conjunto(n))
	ent Terror;
	t1 := CreaVarTemp();
	emite (id_1.pos ':= 0');
	I.fin := Crealans(SIGNINST);
	I.inicio := SIGINST;
	emite('if' id_1.pos '>' buscalimite(id_2.nom) 'goto' __);
	emite (t1 ':=' id_2.pos '[' id_1.pos ']') ;
	salta := Crealans(SINGINST);
	emite('if' t1 '= 0 goto' __); }
	
	{ completalans(salta, SINGINST);
	emite(id_1.pos ':=' id_1.pos '+ 1');
	emite('goto' I.inicio);
	completalans(I.fin, si); }