Universidad
Europea de Madrid

LAUREATE INTERNATIONAL UNIVERSITIES

GENERACIÓN DE CÓDIGO INTERMEDIO

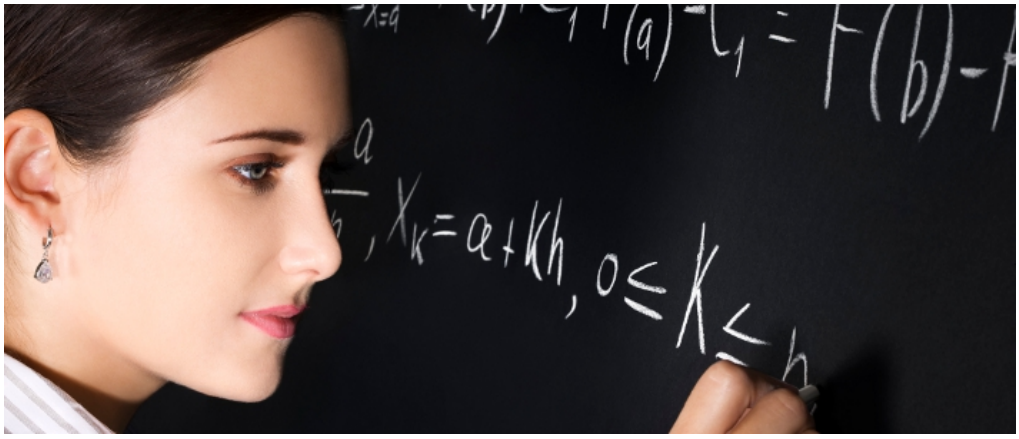
EJEMPLOS PARA DISTINTAS ESTRUCTURAS DE DATOS

Índice

Presentación

El objetivo de este tema es comprender cómo aplicar la generación de código intermedio, mediante el código de tres direcciones a determinadas proposiciones. Para todo ello se alcanzarán los siguientes objetivos:

- Comprender cuáles son las funciones de las expresiones booleanas, así como sus formas de codificación.
- Obtener una *Definición Dirigida por la Sintaxis (DDS)* razonada para las expresiones booleanas.
- Conocer las proposiciones de control de flujo.
- Obtener una DDS para las proposiciones de control de flujo más importantes: **if-then, if-then-else y while.**
- Entender la problemática del relleno de los valores de las etiquetas y su solución.
- Comprender el funcionamiento de las **llamadas a funciones y procedimientos.**



Expresiones booleanas

Las funciones principales de las expresiones booleanas son:

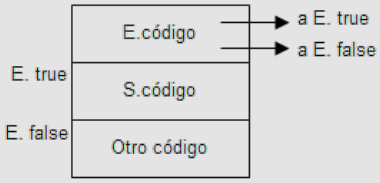
- Cálculo de valores lógicos.
- Expresiones de condición: se utilizan en las proposiciones de control de flujo (if-then-else, while, for, etc.).

Los operadores que se aplican en este tipo de expresiones se denominan **operadores booleanos** y estos son: and, or y not, que en C y C# se escriben como , || y !, respectivamente. Estos operadores booleanos se aplican a variables booleanas.

Expresiones relacionales y aritméticas

Hay que tener en cuenta cómo vamos a representar los valores de una expresión booleana. Por parte de Aho *et al* (1986) se proponen dos métodos:

Codificación numérica de true y false	Codificar numéricamente los valores true y false y evaluar una expresión booleana igual que una aritmética. Podemos asignar a *true* el valor 1 y a *false* el valor 2 y ver el siguiente ejemplo: if (x > y) then a = 3 50: if x > y goto 53 51: t:= 0 52: goto 55 53: t:=1 54: a:= 3 55: ....

Control de flujo	Consiste en representar el valor de una expresión booleana mediante una posición alcanzada en un programa. Este método es muy útil para expresiones booleanas en proposiciones de control como *if-then-else* o *while*. Supongamos el caso del *if-then*, que se representaría así: 

En detalle

Expresiones relacionales y aritméticas

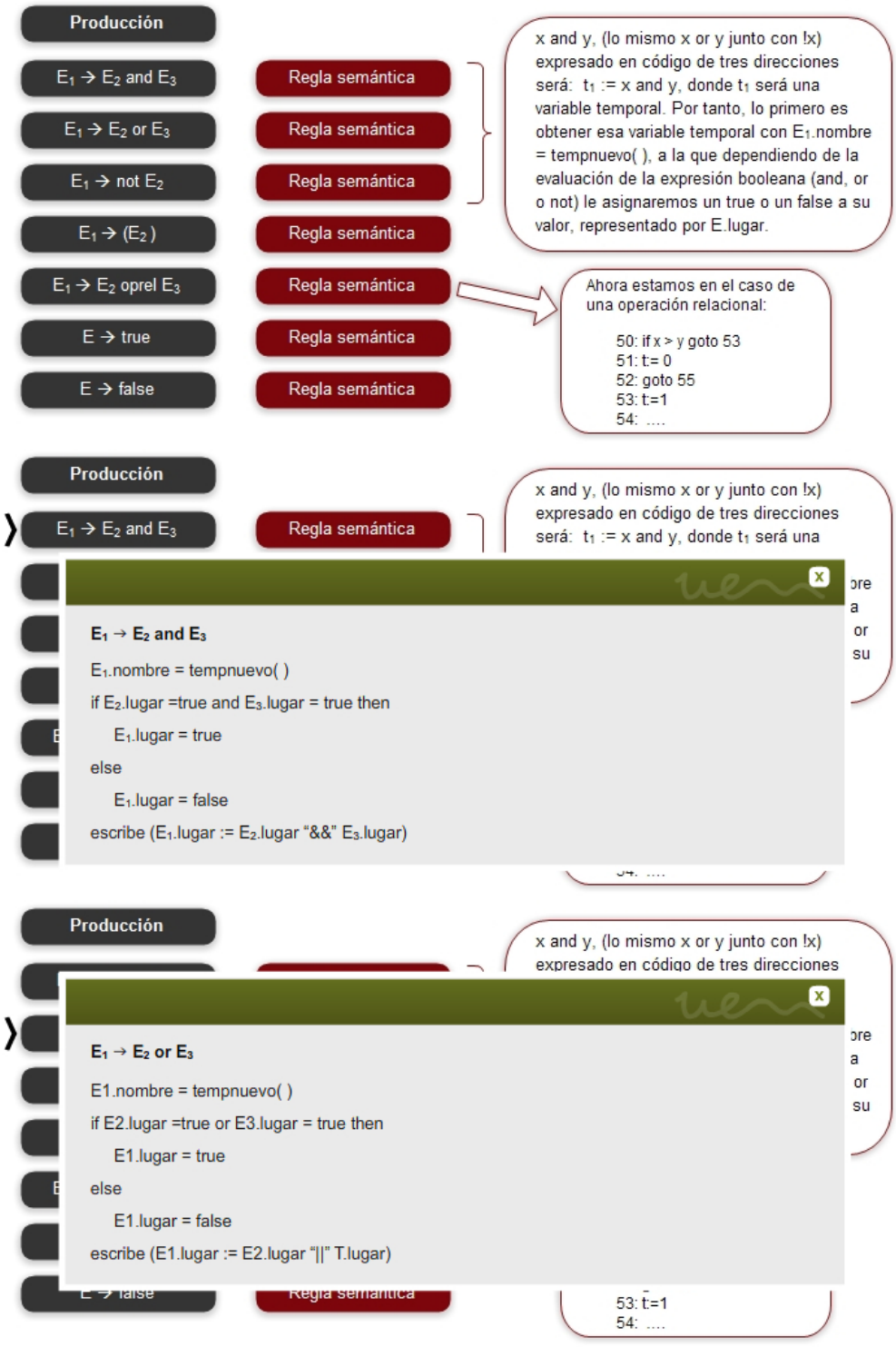
También se pueden construir expresiones relacionales (E_1 oprel E_2), cuyo resultado será un valor lógico (true o false), que en este contexto además serán los atributos que utilizaremos para evaluar estas expresiones. Estas expresiones relacionales usan los siguientes operadores relacionales: >, >=, <, <=, == y !=. Estos a su vez se aplican sobre expresiones aritméticas (E_1 y E_2), que finalmente pueden ser identificadores o números (ej: a > 3).

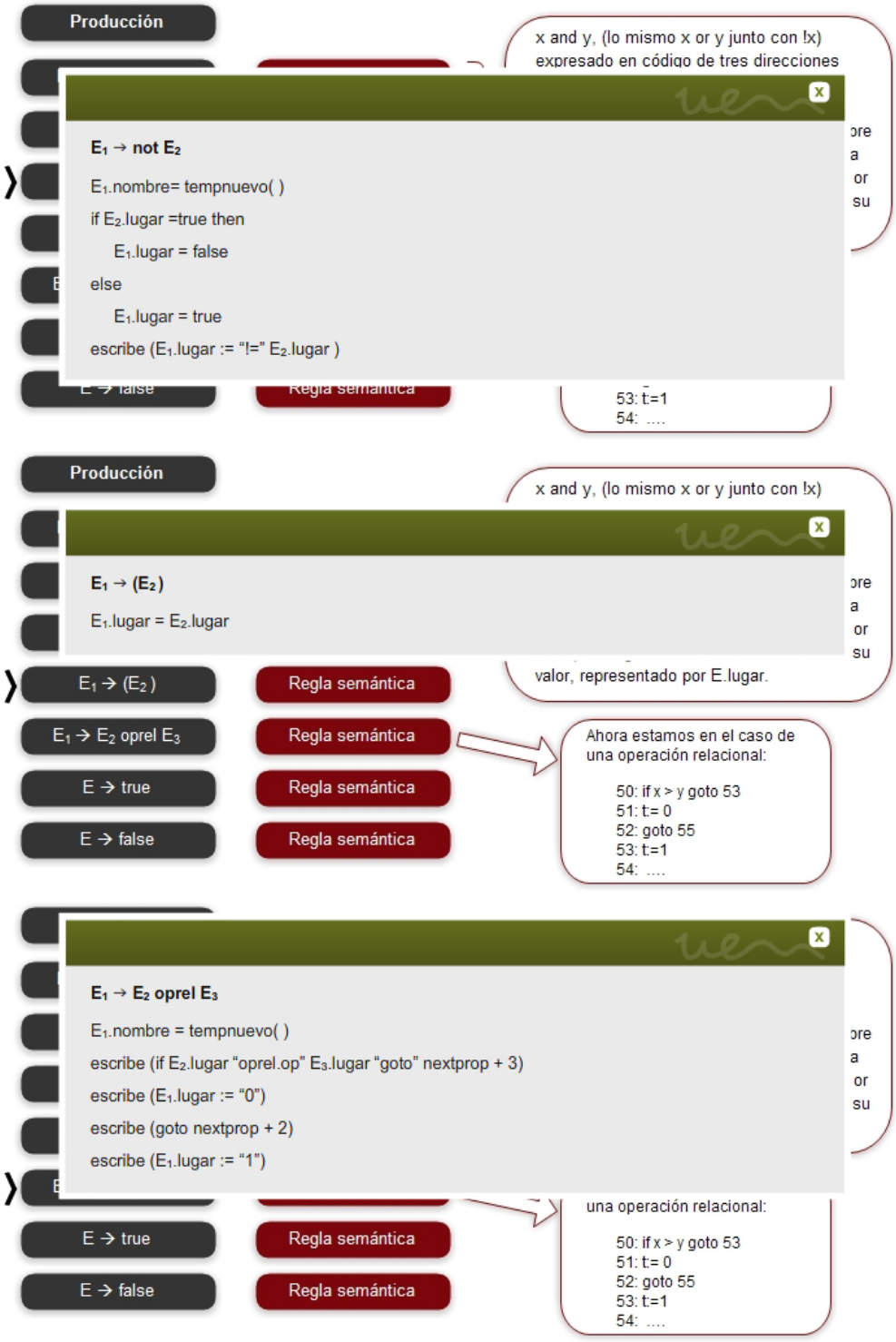
DDS para expresiones booleanas

La gramática para las expresiones booleanas es la siguiente:

E → E and E | E or E |not E | E oprel E | true | false

Utilizamos los siguientes atributos sintetizados.





Documentos DDS para expresiones booleanas

Atributos sintetizados

Utilizamos los siguientes atributos sintetizados:

- **E.nombre:** es el nombre que tendrá la variable temporal.
- **E.lugar:** es el nombre que tendrá el valor lógico de E (true o false).
- **Función *tempnuevo():*** devolverá el nombre del siguiente temporal.
- **Función *escribe( )*:** va a escribir en un fichero el código intermedio que se va generando.

Vamos a usar nextprop para tener el índice de la siguiente proposición.

Proposiciones de control de flujo

Las proposiciones de control de flujo se distinguen por la necesidad de generar etiquetas a las que saltará la secuencia del programa según se cumpla o no la condición o expresión booleana. Esta generación de etiquetas provoca un problema debido a que en el momento en el que se generan no se sabe dónde van a estar ubicadas.

Además de las proposiciones de control como el **if-then, if-then-else o while,** tenemos el bucle **for,** la proposición **case,** o **repeat,** las cuales son muy parecidas a estas tres que definimos.

Las principales proposiciones de control de flujo son:

- if E then S.
- if E then S else S.
- while E do S.

E es la expresión booleana que hay que evaluar y convertir a código de tres direcciones. Utilizaremos una función *nuevaetiq( )* que nos devolverá una nueva etiqueta cada vez que se llama. Estas etiquetas serán *E.true* y *E.false,* y es donde irá el control de flujo una vez se evalúe la mencionada expresión booleana.

En detalle Atributo *código*

Como nos indica Aho *et al* (2008) este *código* pude generarse directamente, sin necesidad de generar el árbol sintáctico y retrasar esta generación de código cuando se recorra el árbol.

También necesitamos un atributo heredado denominado *siguiente,* que nos proporciona el valor donde empieza la primera instrucción después del código correspondiente a S.

En detalle

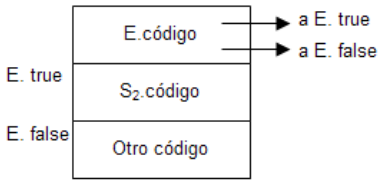
Atributo *código*

También vamos a necesitar otro atributo denominado código (E.código), que será el que tiene el código de tres direcciones de las instrucciones, siendo por tanto del tipo cadena.

DDS para if-then

La generación de código para las sentencias *if-then* e *if-then-else* pasa por entender a través de un esquema la forma en la que este se genera.

Empecemos por el de la sentencia **S_1 → if E then S_2:**



El patrón de código para *if (x > y) then a = 3* será:

```
t₁ = true o false (dependiendo de la evaluación de x > y).
If t₁ = false goto Etiq1
   Código para S₂ (a:= 3)
Etiq1
  Otro código
```

En detalle Implementación en forma de regla semántica

Hay que tener en cuenta que las expresiones booleanas y sus reglas semánticas complementan a las proposiciones de control de flujo en el proceso de evaluación de la condición.

En detalle

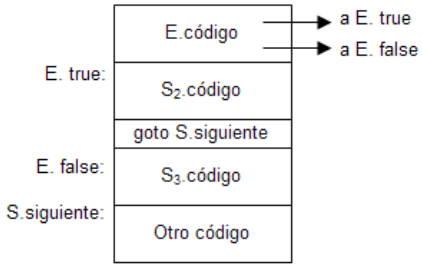
Implementación en forma de regla semántica

Su implementación en forma de regla semántica será:

Producción	S_1 → if E then S_2
Regla semántica	E.true := nuevaetiq( ) E.false := S_1.siguiente S_2.siguiente := S_1.siguiente S_1.código = E.código || escribe(E.true:) || S_2.código

DDS para if-then-else

Como en el caso anterior, se tiene que partir del esquema para **S_1 →if E then S_2 else S_3:**



El patrón de código para *if (x > y) then a = 3 else a = 2* será:

```
t1 = true o false (dependiendo de la evaluación de x > y).
If t1 = false goto Etiq1
   Código para S2 (a:= 3)
   Goto Etiq2
Etiq1
   Código para S3 (a:= 2)
Etiq2
   Otro código
```

Implementación en forma de regla semántica

En detalle

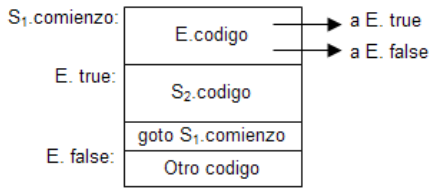
Implementación en forma de regla semántica

Su implementación en forma de regla semántica será:

Producción	$S_1 \rightarrow$ if E then S_2 else S_3
Regla semántica	E.true := nuevaetiq( ) E.false := nuevaetiq( ) S_2.siguiente := S_1.siguiente S_3.siguiente := S_1.siguiente S_1.código = E.código || escribe(E.true:)||S_2.código|| escribe ("goto" S_1.siguiente|| escribe(E.false:) || S_3.código

DDS para while

Realizamos el esquema para $S_1 \rightarrow$ while E do S_2:



El patrón de código para *while (x > y) do a = a +1* será:

```
Etiq1
t1 = true o false (dependiendo de la evaluación de x > y).
If t1 = false goto Etiq2
   Código para S2 (a:= a + 1)
   goto Etiq1
Etiq2
   Otro código
```

Su implementación en forma de regla semántica será:

Implementación
En detalle

En detalle

Implementación

Implementación en forma de regla semántica:

Producción	$S_1 \rightarrow$ while E do S_2
Regla semántica	S_1.comienzo := nuevaetiq( ) E.true := nuevaetiq( ) E.false := S_1.siguiente S_2.siguiente := S_1.comienzo S_1.código = escribe (S_1.comienzo:) || E.código || escribe(E.true:) || S_2.código escribe("goto" S_1.comienzo)

Relleno de etiquetas por retroceso

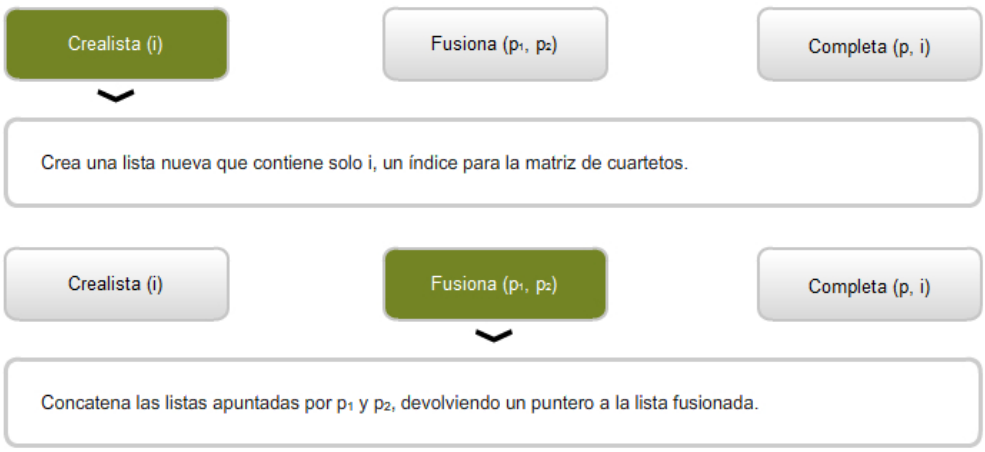
Como se indicó anteriormente, el principal problema de las proposiciones de control de flujo es rellenar estas etiquetas con un valor que todavía no conocemos, puesto que hasta que no está todo el código no se sabe dónde irán los saltos. Este problema se evita de una forma sencilla, aunque poco eficiente, dando dos pasadas.

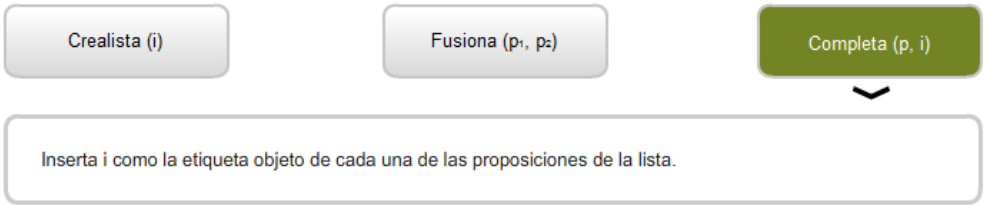
Una solución más eficiente a este problema (Aho *et al,* 1986) consiste en generar las proposiciones de bifurcación, pero sin especificar los destinos de los saltos, e ir completando una **lista de proposiciones *goto,*** cuyas etiquetas se rellenarán cuando se pueda determinar la etiqueta adecuada. Esto se denomina **relleno por retroceso.**

Esto requiere que el código generado se mantenga en un buffer, de manera que se pueda rellenar mediante una única pasada.

Louden (2004) indica que el proceso de relleno puede generar otros problemas debido a las arquitecturas de las máquinas, puesto que muchas permiten dos variedades de saltos: un salto corto (dentro del alcance de 128 bytes), y un salto largo que requiere más espacio de código.

Para manejar las listas de etiquetas, Aho *et al* (1986) proponen tres funciones:





En Aho *et al* (1986 y 2008) hay ejemplos de implementación incluyendo las reglas semánticas necesarias.

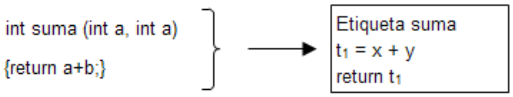
Llamadas a funciones

Las tareas más importantes en las funciones o procedimiento son la gestión de las llamadas y de los retornos. Para poder hacer la llamada a una función antes se tiene que haber realizado la declaración o definición de la misma.

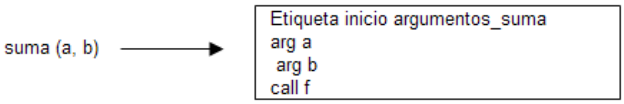
Hay que recordar que la arquitectura de memoria de la máquina es importante en este proceso, puesto que interviene en la forma en la que se reserva el espacio para la función, así como para tener control del valor devuelto por la función o procedimiento. Por todo ello, las tareas a realizar, aparte de la generación del código de la función en cuestión, son:

Declaración de la función	Esto implica introducir en la tabla de símbolos el nombre de la función, el tipo que devuelve y los parámetros o argumentos de la función. Ejemplos de declaración de funciones son los siguientes: *int nuevatemporal( )*, o bien *boolean expresión(int a, char oprel, int b).*
Llamada a la función	1. Secuencia de llamada: incluir espacio para el registro de activación, argumentos y variables locales (incluyendo las variables temporales). 2. Secuencia de retorno: ubicación del valor de retorno, liberación de registros y liberación del espacio ocupado por el registro de activación.

El código de tres direcciones para la declaración de una función sencilla como:



Para la llamada a la función hay que indicar el inicio del cálculo del argumento, los argumentos para los que utilizamos la instrucción arg y, por último, la instrucción de llamada a funciones que llamamos call. Un ejemplo sería:



En Louden (2004) se define una gramática y algunos ejemplos de implementación.

Memoria

La forma en la que se gestiona la asignación de memoria para una función se basa en lo que se denomina **registro de activación** (Louden, 2004), que es donde se almacenan los datos locales de una función o procedimiento. Un registro de activación contendrá como mínimo: espacio para los argumentos, espacio para las direcciones de retorno, espacio de datos locales de la función y espacio para temporales locales de la función.

Resumen

En este tema hemos aprendido cómo se construye el **código de tres direcciones** para las expresiones booleanas y las de control de flujo.

En primer lugar, se han definido los esquemas necesarios para entender el proceso, así como el patrón de código correspondiente al mismo, con el objeto de que el código que se genera mediante las reglas semánticas sea totalmente trazable y entendible.

También se ha visto **cómo obtener variables temporales para los cálculos y las etiquetas para los saltos de flujo.** Estos saltos se realizan una vez se ha evaluado la condición o expresión booleana correspondiente y hay que obtener el valor de la dirección de memoria a la que se va a saltar. Para resolver este problema, se ha visto una solución denominada **relleno por retroceso.**

Por último se han identificado las tareas necesarias para generar código de tres direcciones para las llamadas a funciones y las declaraciones de las mismas.