4º Ingeniería Informática

II26 Procesadores de lenguaje

Generación de código

Esquema del tema

1. Introducción

Una vez ha terminado el análisis y se ha obtenido un AST decorado, comienza la generación de código. Esta fase suele dividirse en dos partes: generación de código intermedio y generación de código de máquina.

El código intermedio se genera para una máquina virtual. Estas máquinas se definen con dos objetivos:

- Ser lo suficientemente simples como para poder generar código para ellas de manera sencilla, pero con la suficiente riqueza para poder expresar las construcciones del lenguaje fuente.
- Estar lo suficientemente próximas a los procesadores reales para que el paso del código intermedio al código de máquina sea prácticamente directo.

Para conseguirlo, las máquinas virtuales tienen una serie de características que las hacen simples. Por ejemplo:

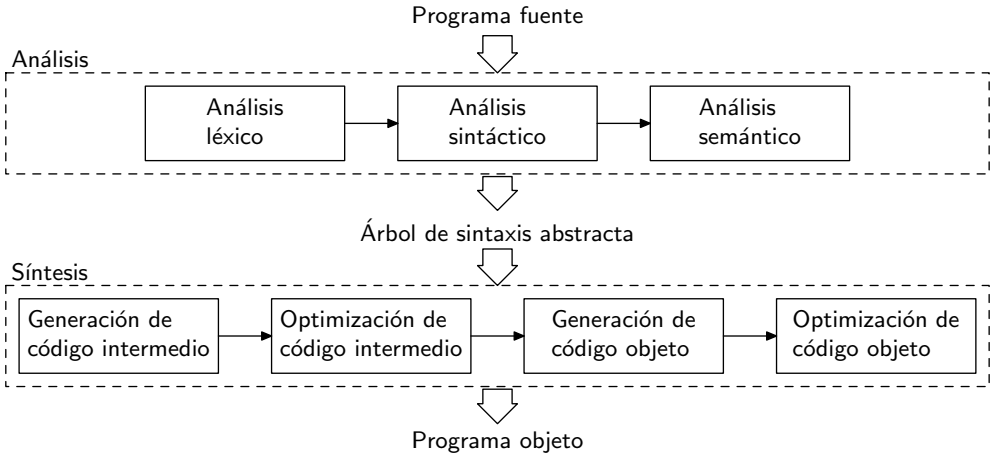
- Tienen pocos modos de direccionamiento.
- Suelen tener un número ilimitado de registros (que pueden estar organizados en forma de pila).
- Tienen un juego de instrucciones relativamente simple.

En principio, el código intermedio es independiente del lenguaje de programación fuente. En la práctica, son habituales los códigos intermedios que facilitan la representación de determinadas características de un lenguaje concreto. Así, el *P-code* está pensado para traducir Pascal y el *Java-bytecode* es muy bueno para el Java.

Otra de las ventajas del código intermedio es que facilita la escritura de compiladores para distintas máquinas. La traducción del lenguaje a código intermedio sería idéntica en todas y lo único que cambiaría sería el traductor de código intermedio a código de máquina. Esta idea de un código intermedio independiente de la máquina puede llevarse un paso más allá, como se hace en diversos lenguajes, de los cuales Java es quizá el más representativo. La idea es compilar el lenguaje a un código intermedio. Al ejecutar el programa, se interpreta este código intermedio o se compila "al vuelo" para la máquina destino. De esta manera, un mismo "binario" puede ejecutarse en máquinas con arquitecturas y sistemas operativos totalmente distintos.

En el modelo clásico, que seguiremos, una vez generado el código intermedio, se traduce éste a código de máquina. La traducción tiene que hacerse de manera que se aprovechen adecuadamente los recursos de la máquina real.

Aunque en principio estas dos fases bastan para la generación del código final, en la práctica están mezcladas con fases de optimización. Es habitual tener entonces un esquema similar al que vimos en el primer tema:



Podríamos entonces decir que la generación de código tiene como objetivo generar el ejecutable que después empleará el usuario. Sin embargo, es habitual que el producto del compilador no sea directamente un fichero ejecutable. Es bastante más común que sea un fichero en lenguaje ensamblador. De esta manera, se evitan problemas como tener que medir el tamaño exacto de las instrucciones o llevar la cuenta de sus direcciones. Además, es muy probable que el código generado tenga referencias a objetos externos como funciones de biblioteca. Estas referencias externas serán resueltas por el enlazador o el cargador.

1.1. Lo más fácil: no generar código

Dado que la generación de código es una tarea compleja, especialmente si queremos que sea eficiente y razonablemente compacto, es interesante buscar maneras de evitar esta fase. De hecho existe una manera sencilla de evitar la generación de código (al menos, la generación de código máquina) que permite obtener ejecutables de muy buena calidad: generar código en otro lenguaje para el que haya un buen compilador.

Hoy en día, prácticamente cualquier sistema tiene compiladores de C de gran calidad. Es posible, por tanto, traducir nuestro lenguaje a C y después compilar el programa obtenido con un buen compilador. Esto tiene varias ventajas:

- El tiempo de desarrollo de nuestro compilador se reduce considerablemente.
- En muchos sistemas, el compilador de C genera código de muy alta calidad ya que de él dependen muchos programas críticos (entre ellos el propio sistema operativo).
- Nos permite tener un compilador para gran número de máquinas con un esfuerzo mínimo (el de adaptarse a las pequeñas diferencias entre unos compiladores y otros). De hecho, en algunos casos se habla del lenguaje C como de un "ensamblador independiente de la máquina".

En el caso de lenguajes de muy alto nivel o de paradigmas como el funcional y el lógico, esta es prácticamente la elección unánime. Hay incluso propuestas de lenguajes como el C--, que buscan un lenguaje reducido y con ciertas características de más bajo nivel como lenguaje destino para muchos compiladores.

Si hemos construido el árbol mediante clases, la traducción a C es prácticamente trivial para muchas de las construcciones habituales en los lenguajes imperativos. Por ejemplo, para los nodos correspondientes a las expresiones aritméticas, podemos utilizar algo parecido a:

Objeto NodoSuma:
    ...
    Método traduceaC():
        devuelve "(" + izdo.traduceaC() + "+" + dcho.traduceaC() + ")";
    fin traduceaC
    ...
fin NodoSuma

donde izdo y dcho son los hijos izquierdo y derecho, respectivamente.

Igual que vimos en el tema de análisis semántico, tanto en este caso como en los que comentemos después, no es necesario que utilicemos una clase por cada tipo de nodo. Tampoco es necesario que utilicemos objetos y métodos: podemos recorrer el árbol con ayuda de una pila o de funciones recursivas adecuadas.

2. Código intermedio

Existen códigos intermedios de diversos tipos que varían en cuanto a su sencillez, lo próximos que están a las máquinas reales y lo fácil que es trabajar con ellos. Nosotros nos centraremos en un tipo de código que se parece bastante al lenguaje ensamblador. Existen otros tipos de código intermedio que representan los programas como árboles o grafos. También existen representaciones mixtas que combinan grafos o árboles y representaciones lineales.

El formato que usaremos para las operaciones binarias es similar al del ensamblador MIPS: *op dst, op1, op2*, donde

- *op* es un operador,
- *dst* es el registro destino de la operación,
- *op1* y *op2* son los operandos. Si *op* termina en `i`, el segundo operando es un valor inmediato (entero o real).

En caso de que el operador sea unario, la forma de la instrucción es *op dst, op1*. Para acceder a memoria utilizaremos instrucciones de acceso a memoria usando un registro base y un desplazamiento. Para leer de memoria utilizamos `lw` *dst, desp(base)* donde *dst* es el registro destino, *desp* es un entero que representa el desplazamiento y *base* es el registro base. Si queremos acceder a una dirección absoluta, utilizamos `$zero` como registro base, aprovechando que este registro siempre devuelve el valor 0. Para escribir en memoria utilizamos `sw` *fnt, desp(base)* donde *fnt* es el registro fuente y el resto tiene el mismo significado que antes.

Por ejemplo, la sentencia `a:= b*(-c)`, donde `a` es una variable local en la dirección 1 respecto al registro `$fp` y `b` y `c` son variables globales en las direcciones 1000 y 1001, se puede traducir como:

```
lw $r1, 1000($zero)
lw $r2, 1001($zero)
multi $r3, $r2, -1
mult $r4, $r1, $r3
sw $r4, 1($fp)
```

Uno de los objetivos que suele perseguirse es reducir al mínimo el número de registros utilizados. En nuestro caso, podemos emplear dos:

```
lw $r1, 1000($zero)
lw $r2, 1001($zero)
multi $r2, $r2, -1
mult $r1, $r1, $r2
sw $r1, 1($fp)
```

Si las variables estuvieran en los registros[1] `$r1`, `$r2` y `$r3`, podríamos incluso no utilizar ningún registro auxiliar:

```
multi $r1, $r3, -1
mult $r1, $r2, $r1
```

También tendremos instrucciones para controlar el flujo de ejecución, para gestionar entrada-salida, etc. A lo largo del tema iremos viendo esas instrucciones según las vayamos necesitando.

3. Generación de código para expresiones

Empezaremos por generar código para las expresiones. Asumiremos que para gestionar el código disponemos de una función auxiliar, *emite*. Esta función recibe una cadena que representa una línea de código intermedio y hace lo necesario para que termine en el programa final; por ejemplo, puede escribirla en un fichero, almacenarla en una lista para pasarla a otro módulo, etc.

3.1. Expresiones aritméticas

Comenzamos el estudio por las expresiones aritméticas. Lo que tendremos que hacer es crear por cada tipo de nodo un método, generaCódigo, que genere el código para calcular la expresión y lo emita. Ese código dejará el resultado en un registro, cuyo nombre devolverá generaCódigo como resultado.

Para reservar estos registros temporales, utilizaremos una función, *reservaReg*. En principio bastará con que esta función devuelva un registro distinto cada vez que se la llame.

Cada nodo generará el código de la siguiente manera:

- Por cada uno de sus operandos, llamará al método correspondiente para que se evalúe la subexpresión.
- Si es necesario, reservará un registro para guardar su resultado.
- Emitirá las instrucciones necesarias para realizar el cálculo a partir de los operandos.

Con este esquema, la generación del código para una suma será:

Objeto NodoSuma:
...
Método generaCódigo()
izda:= *i*.generaCódigo();
dcha:= *d*.generaCódigo();
r :=*reservaReg*();
emite(`add` *r*, izda, dcha);
devuelve *r*;
fin generaCódigo
...
fin NodoSuma

En el caso de los operadores unarios, bastará con reservar un registro y hacer la correspondiente operación.

Ejercicio 1

Escribe el método de generación de código para el operador unario de cambio de signo.

[1] En principio sería raro que las variables globales residieran en registros pero no sería tan raro para la variable local.

La generación de código para constantes se limita a almacenar el valor en un registro:

Objeto NodoConstante:
    ...
    Método generaCódigo()
        r :=*reservaReg*();
        emite(`addi` r`, $zero,` valor);
        devuelve r;
    fin generaCódigo
    ...
fin NodoConstante

Para las variables, tendremos que distinguir según si son locales o globales y acceder a la dirección correspondiente. Las variables globales tendrán una dirección absoluta a la que accederemos utilizando el registro `$zero` como base y las variables locales y parámetros de las funciones tendrán una dirección relativa al registro `$fp` (*frame pointer*):

Objeto NodoAccesoVariable:
    ...
    Método generaCódigo()
        r :=*reservaReg*();
        if eslocal **entonces**
            emite(`lw` r, dir`($fp)`);
        si no
            emite(`lw` r, dir`($zero)`);
        fin si
        devuelve r;
    fin generaCódigo
    ...
fin NodoVariableGlobal

Observa que asumimos que las variables están en memoria y no permanentemente en registros. Esto no es óptimo, pero nos permite presentar un método de generación razonablemente sencillo.

Por ejemplo, vamos a traducir la expresión `(a+2)*(b+c)`. Si suponemos que las variables están en las direcciones 1000, 1001 y 1002, respectivamente, el código que se generará es:

```
lw $r1, 1000($zero)
addi $r2, $zero, 2
add $r3, $r1, $r2
lw $r4, 1001($zero)
lw $r5, 1002($zero)
add $r6, $r4, $r5
mult $r7, $r3, $r6
```

Es bueno intentar reducir el número de registros utilizados. Por ejemplo, al llamar a una función, hay que guardar en la trama de activación todos los registros activos en ese punto. Por eso, será bueno intentar "reutilizar" los registros y tener el mínimo número de ellos ocupados. Si vamos a tener una fase de optimización, no hace falta ocuparse ahora de esta cuestión. Si no tenemos un optimizador, podemos utilizar una estrategia sencilla que reduce bastante el número de registros empleados. La idea es tener también una función *liberaReg* que marca un registro como disponible para una nueva llamada de *reservaReg*. Para esto, podemos tener una lista de registros activos y otra de registros libres. La llamada a *reservaReg* devuelve un elemento de la lista de registros libres, si no está vacía, o crea un nuevo registro si lo está. La llamada a *liberaReg* mueve el registro a la lista de registros libres. La lista de registros activos se emplea, por ejemplo, para saber qué registros hay que guardar en las llamadas a función[2].

[2]Más adelante veremos que cuando se llama a una función es necesario guardar aquellos registros que contienen algún valor que interesa preservar.

Siguiendo esta idea, el nodo de la suma sería:

Objeto NodoSuma:
    ...
    Método generaCódigo()
        izda:= *i*.generaCódigo();
        dcha:= *d*.generaCódigo();
        emite(`add` *r*, izda, dcha);
        liberaReg(dcha);
        devuelve izda;
    fin generaCódigo
    ...
fin NodoSuma

Para `(a+2)*(b+c)` se generaría:

```
lw $r1, 1000($zero)
addi $r2, $zero, 2
add $r1, $r1, $r2
lw $r2, 1001($zero)
lw $r3, 1002($zero)
add $r2, $r2, $r3
mult $r1, $r1, $r2
```

En el caso de los unarios, directamente reutilizaremos el registro correspondiente al operando.

Ejercicio 2

Escribe el método de generación de código para el operador unario de cambio de signo reutilizando registros.

¿Necesitas tener una clase por cada operador del lenguaje? La respuesta dependerá principalmente de cómo te sientas más cómodo. En principio, tienes un rango de posibilidades, desde una clase por operador a una única clase que tenga como atributo el operador concreto. Un compromiso es tener clases que agrupen operadores con propiedades similares, como por ejemplo los lógicos por un lado y los aritméticos por otro. Lo único que hay que tener en cuenta en caso de que haya más de un operador representado en una clase es que pueden existir sutilezas que los distingan. Por ejemplo, en Pascal, el operador `/` siempre devuelve un real, mientras que el tipo del operador `+` depende de los de sus operandos.

3.2. Coerción de tipos

Antes hemos supuesto implícitamente que sólo teníamos un tipo de datos con el que operábamos. En la práctica es habitual tener, al menos, un tipo de datos entero y otro flotante. En este caso, es necesario generar las operaciones del tipo adecuado y, si es necesario, convertir los operandos. Supondremos que nuestro código intermedio tiene un operador unario, `tofloat`, para transformar su operando entero en un número real. Además, contamos con las versiones reales de los operadores, que se distinguen por tener una "`f`" delante. La generación de código para la suma en este caso es la mostrada en la figura 1.

Al aumentar el número de tipos disponibles o si se usan registros distintos para tipos distintos el código se complica rápidamente. Suele ser preferible añadir la conversión de tipos de forma

```
Objeto NodoSuma:
    ...
    Método generaCódigo()
        izda:= i.generaCódigo();
        si i.tipo≠ tipo entonces
            izdaR:= reservaRegReal();
            emite(tofloat izdaR, izda);
            liberaReg(izda);
            izda:= izdaR
        fin si
        dcha:=d.generaCódigo();
        si d.tipo≠ tipo entonces
            dchaR:= reservaRegReal();
            emite(tofloat dchaR, dcha);
            liberaReg(dcha);
            dcha:= dchaR
        fin si
        si tipo=real entonces
            emite(fadd izda, izda, dcha)
            liberaRegReal(dcha);
        si no
            emite(add izda, izda, dcha)
            liberaReg(dcha);
        fin si
        devuelve izda;
    fin generaCódigo
    ...
fin NodoSuma
```

Figura 1: Generación de código para las sumas.

explícita en el AST. Por ejemplo, un nodo para convertir de entero a real sería:

```
Objeto NodoEnteroAReal:
    ...
    Método generaCódigo():
        r:= exp.generaCódigo()
        rR:= reservaRegReal();
        emite(tofloat rR, r);
        liberaReg(r);
        devuelve rR;
    fin generaCódigo
    ...
fin NodoEnteroAReal
```

Ejercicio 3

Escribe el método de generación de código para la suma suponiendo que hay nodos explícitos para la conversión de tipos.

3.3. Asignaciones

En algunos lenguajes de programación, la asignación es un operador más, mientras que en otros se la considera una sentencia. El tratamiento en ambos casos es muy similar. La diferencia es que

en el primer caso se devuelve un valor, generalmente el mismo que se asigna a la parte izquierda, mientras que, en el segundo caso, lo importante de la asignación es su efecto secundario.

Si la parte izquierda de la asignación puede ser únicamente una variable simple, basta con generar el código de modo que se calcule la expresión de la parte derecha y se almacene el resultado en la dirección de la variable:

Objeto NodoAsignación:
...
Método generaCódigo()
dcha:= *d*.generaCódigo();
si eslocal **entonces**
emite(`sw` dcha, dir`($fp)`);
si no
emite(`sw` dcha, dir`($zero)`);
fin si
devuelve dcha
fin generaCódigo
...
fin NodoAsignación

Por claridad, hemos omitido la comprobación de tipos. Si consideramos la asignación como sentencia, quitamos la devolución de dcha y lo sustituimos por *liberaReg*(dcha).

Si podemos tener en la parte izquierda variables estructuradas, tenemos dos alternativas: podemos tener un nodo por cada posible estructura (NodoAsignaVector, NodoAsignaRegistro, etc.) o mantener un único nodo. Para hacerlo, añadiremos a aquellos nodos que puedan aparecer en una parte izquierda un nuevo método, generaDir, que genera la dirección del objeto correspondiente y la devuelve en un registro. De esta manera, lo que tiene que hacer la asignación es generar código para la expresión, generar la dirección de la parte izquierda y hacer la asignación:

Objeto NodoAsignación:
...
Método generaCódigo()
dcha:= *d*.generaCódigo();
dir:= *i*.generaDir();
emite(`sw` dcha, `0(`dir`)`);
liberaReg(izda);
devuelve dcha;
fin generaCódigo
...
fin NodoAsignación

Ejercicio 4

Escribe el método de generación de dirección para las variables simples. Ten en cuenta que las variables pueden ser locales o globales.

Ejercicio 5

¿Qué código se generaría para la sentencia `a:= b*d/(e+f)`? Supón que `a` y `b` son variables globales de tipo entero con dirección 100 y 200, respectivamente, y el resto locales con desplazamientos 1, 2 y 3.

3.4. Vectores

Para acceder a vectores, asumiremos que sus elementos ocupan posiciones consecutivas en la memoria. Esto quiere decir que un vector como, por ejemplo, `a: ARRAY [2..5] OF integer` se

almacena en memoria de la siguiente forma:

a[2]	a[3]	a[4]	a[5]

Supongamos que el vector comienza en la dirección base b y cada nodo tiene un tamaño t. Llamemos l al límite inferior (en nuestro caso $l = 2$). Para acceder al elemento i, podemos utilizar la fórmula:

$$d = b + (i - l) \times t$$

Esta dirección se obtiene sumando a la dirección base el número de elementos delante del que buscamos $(i - l)$ multiplicado por el tamaño de cada elemento (t). En general, el compilador deberá emitir código para hacer ese cálculo en tiempo de ejecución. Podemos ahorrar tiempo de ejecución si reescribimos la fórmula anterior como sigue:

$$d = i \times t + (b - l \times t)$$

Esta expresión se puede interpretar como la suma del desplazamiento correspondiente a i elementos $(i \times t)$ con la dirección del elemento cero del vector, que es $b - l \times t$ y se puede calcular en tiempo de compilación.

Si llamamos "base virtual" a la dirección del elemento cero, la generación de la dirección queda:

Objeto NodoAccesoVector:
    ...
    Método generaDir()
        d := *exp*.generaCódigo(); // Desplazamiento
        emite(`mult` d`,` d`,` t);
        emite(`add` d`,` d`,` base_virtual);
        devuelve d;
    fin generaCódigo
    ...
fin NodoAccesoVector

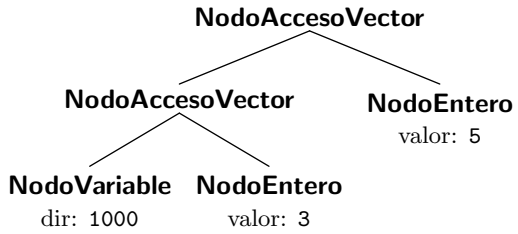
Ejercicio 6

Escribe el código generado para la sentencia `a[i]:= 5;`. Supón que `a` es de tipo `ARRAY[1..5] OF integer`, que los enteros ocupan cuatro bytes y que `a[1]` está en la dirección 1000 e `i` en la 1100.

Ejercicio 7

Escribe el método de lectura de un elemento de un vector. Puedes aprovechar el método generaDir.

Si el vector es n-dimensional, resulta cómodo interpretarlo como un vector unidimensional de vectores $n - 1$ dimensionales y representar los accesos al vector como anidamientos de accesos unidimensionales. Podemos, por ejemplo, representar `a[3][5]` como:



Esto supone cambiar NodoAccesoVector para tener en cuenta que la base virtual debe incluir los límites inferiores de cada dimensión. Para simplificar la exposición, asumiremos que, como en C, el límite inferior es cero. En ese caso, el cálculo de la dirección queda:

Objeto NodoAccesoVector:
...
Método generaDir()
r :=izda.generaDir(); // Base del vector
d :=exp.generaCódigo(); // Desplazamiento
emite(`mult` d`,` d`,` t);
emite(`add` r`,` r`,` d);
liberaReg(d);
devuelve r;
fin generaDir
...
fin NodoAccesoVector

Ejercicio 8

Escribe el código generado para la sentencia C `a[3][5]=25;` suponiendo que se ha declarado `a` de tipo `int a[10][10]`, que está en la dirección 1000 y que los enteros ocupan cuatro bytes.

Ejercicio* 9

Escribe el código para el acceso a vectores suponiendo que cada dimensión puede tener un límite inferior. Intenta reducir los cálculos en tiempo de ejecución.

4. Generación de código para las estructuras de control

Una vez sabemos cómo generar código para las expresiones y asignaciones, vamos a ver cómo podemos generar el código de las estructuras de control.

La estructura de control más simple, la secuencia, consiste simplemente en escribir las distintas sentencias una detrás de otra. En cuanto a las llamadas a función, veremos que habrá que crear para ellas dos estructuras: el prólogo y el epílogo. Las sentencias de su cuerpo no tienen nada especial.

A continuación veremos algunas de las estructuras más comunes de los lenguajes de programación imperativos. Otras estructuras se pueden escribir de forma similar.

4.1. Expresiones lógicas

Antes de comenzar con las estructuras propiamente dichas, vamos a ver cómo se puede generar código para las expresiones de tipo lógico, que emplearemos en las condiciones de esas estructuras.

A la hora de trabajar con expresiones de tipo lógico, hay dos opciones:

- Codificar numéricamente los valores lógicos, por ejemplo 1 para cierto y 0 para falso y tratar las expresiones normalmente con operadores booleanos (`or` *x*`,` *y*`,` *z*, `and` *x*`,` *y*`,` *z*, `not` *x*`,` *y*).
- Representar el valor de la expresión mediante el flujo de ejecución del programa. Si la expresión es cierta, el programa llegará a un sitio y, si es falsa, a otro.

El primer método es relativamente sencillo si disponemos de instrucciones que permitan realizar las operaciones booleanas directamente. Por ejemplo, podemos tener algo parecido a la instrucción `lt` *r1*`,` *r2*`,` *r3* que guarde en *r1* un cero o un uno según los valores de *r2* y *r3*. En la práctica, es habitual que las instrucciones que tengamos sean del tipo `blt` *r1*`,` *r2*`,` *Etq*, que provocan un salto a *Etq* si *r1* es menor que *r2*. En este caso, es más aconsejable utilizar la generación mediante control de flujo.

Ejercicio 10

Escribe la traducción de `a<b O NO a<c`. Supón que las variables son globales en 100, 110 y 120, respectivamente y utiliza las instrucciones siguientes:
`or` *r1*`,` *r2*`,` *r3*, `not` *r1*`,` *r2* y `lt` *r1*`,` *r2*`,` *r3*.

Otra razón por la que la representación mediante control de flujo es especialmente interesante es el empleo de *cortocircuitos*. En muchos lenguajes de programación, la evaluación de una expresión se detiene tan pronto como el valor es conocido. Esto quiere decir que si se sabe que el primer operando de un y-lógico es falso, no se evalúa el segundo operando. Análogamente, no se evalúa el segundo operando de un o-lógico si el primero es cierto. Además, el código generado de esta manera se integra perfectamente en las estructuras de control; por ejemplo, al generar el código para una instrucción si, nos interesa que si la condición es cierta, el flujo de ejecución pase a la parte del entonces y si es falsa a la parte del si no.

La generación del código mediante control de flujo se puede hacer añadiendo un método, códigoControl, a cada nodo. Este método recibirá dos parámetros: el primero, cierto, será la etiqueta donde debe saltarse en caso de que la expresión sea cierta; el segundo, falso, será el destino en caso de falsedad. Por ejemplo, para la comparación "menor que", la generación de código sería:

Objeto NodoMenor:
...
Método códigoControl(cierto, falso)
izda:= *i*.generaCódigo();
dcha:= *d*.generaCódigo();
emite(`blt` izda`,` dcha`,` cierto);
emite(`j` falso)
liberaReg(izda);
liberaReg(dcha);
fin códigoControl
...
fin NodoMenor

Como ves, generamos código para los dos hijos. Si la comparación es cierta, saltamos a la etiqueta cierto y, si no, pasamos a un salto incondicional (instrucción `j`) a falso.

En muchas ocasiones el destino del salto es la instrucción siguiente a la comparación. Por ejemplo, en una sentencia si, cuando la condición es cierta hay que saltar a las instrucciones que siguen a la comparación. Nuevamente, la fase de optimización puede eliminar estos saltos superfluos, pero si no tenemos un optimizador separado, podemos mejorar algo nuestro código. Usaremos un valor especial (por ejemplo, `None`) en caso de que queramos que la ejecución vaya a la instrucción siguiente. Asumiremos que los dos parámetros no son `None` simultáneamente.

Con este convenio, el esquema de generación de código para "menor que" es el de la figura 2.

Para el caso en que la ejecución deba pasar a la instrucción siguiente si el resultado es cierto, nos ha bastado con invertir la condición (`bge` salta si su primer registro es mayor o igual que el segundo). Habitualmente, todas las posibles comprobaciones están implementadas en el procesador y esto no supone mayor problema.

Ejercicio 11

Escribe el código generado para la expresión `a<100` para las llamadas siguientes:

- `nodo.códigoControl("et1", "et2")`
- `nodo.códigoControl("et1", None)`
- `nodo.códigoControl(None, "et2")`

Supón que `a` es una variable local con desplazamiento 4 respecto al registro `$fp`.

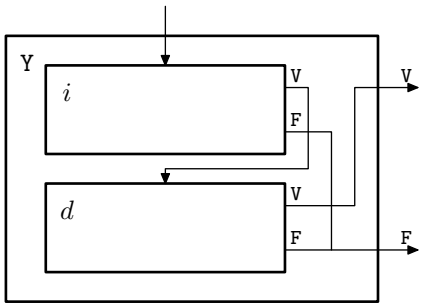
Como en el caso de las operaciones aritméticas, tendremos que tener cuidado con los tipos. Es habitual que tengamos instrucciones separadas para comparar reales y enteros. En ese caso,

```
Objeto NodoMenor:
    ...
    Método códigoControl(cierto, falso)
        izda:= i.generaCódigo();
        dcha:= d.generaCódigo();
        si cierto≠ None entonces
            emite(blt izda, dcha, cierto);
        si no
            emite(bge izda, dcha, falso);
        fin si
        si cierto≠None y falso≠None entonces
            emite(j falso)
        fin si
        liberaReg(izda);
        liberaReg(dcha);
    fin códigoControl
    ...
fin NodoMenor
```

Figura 2: Generación de código para las comparaciones "menor que".

podemos aplicar las mismas estrategias que entonces. De manera similar a como pasaba con las operaciones aritméticas, no es necesario tener exactamente una clase por cada posible comparación; puede bastar con un atributo adicional para indicar el tipo de comparación que se va a hacer.

La generación de código para el y-lógico es también sencilla. Primero, generaremos código para la parte izquierda. Este código tendrá como destino en caso de cierto el código de la parte derecha, que le seguirá. En caso de falso, el control deberá ir al falso del Y. Podemos representarlo gráficamente de la siguiente manera:



Al generar código, tendremos que tener en cuenta que el falso de Y puede ser `None`. Para ese caso, generaremos una nueva etiqueta donde saltar si la parte izquierda resulta ser falsa. El método códigoControl queda como se ve en la figura 3

El caso del o-lógico es análogo.

Ejercicio 12

Escribe el método de generación de código para el o-lógico.

```
Objeto NodoY:
    ...
    Método códigoControl(cierto, falso)
        si falso= None entonces
            aux:=nuevaEtiqueta();
        si no
            aux:=falso;
        fin si
        i.códigoControl(None, aux);
        d.códigoControl(cierto, falso);
        si falso= None entonces
            emite(aux:);
        fin si
    fin códigoControl
    ...
fin NodoY
```

Figura 3: Generación de código para el operador y-lógico.

La generación de código para el no-lógico es extremadamente sencilla, basta con cambiar cierto por falso:

```
Objeto NodoNo:
    ...
    Método códigoControl(cierto, falso)
        e.códigoControl(falso,cierto)
    fin códigoControl
    ...
fin NodoNo
```

Por último, nos queda el problema de cómo generar código para los objetos elementales de tipo lógico. Las constantes cierto y falso son sencillas, basta con un salto incondicional a la etiqueta correspondiente (o nada si es `None`). Para las variables de tipo booleano, hay que decidir una codificación y hacer que el código correspondiente sea la lectura de la dirección de memoria seguida de un salto condicional y, posiblemente, uno incondicional. La asignación a una variable lógica se hará generando el equivalente a dos sentencias, una para asignarle cierto y otra para asignarle el valor falso. El código completo evaluará la expresión, saltando a la instrucción correspondiente según sea necesario.

Ejercicio 13

Escribe los esquemas correspondientes a las constantes lógicas, el uso de una variable de tipo lógico y la asignación a una variable de tipo lógico.

Mención aparte merecen los lenguajes, como C, que no tienen un tipo lógico independiente sino que utilizan los enteros. En este caso, las expresiones aritméticas tienen que tener los dos métodos generaCódigo y códigoControl. El segundo llamará al primero y después generará una instrucción para saltar según el resultado.

Ejercicio 14

Escribe el método códigoControl para la clase NodoSuma en un lenguaje que interprete el cero como falso y cualquier otra cosa como cierto.

Alternativamente, podemos tener un nodo que transforme el valor entero en el control de flujo adecuado. Por ejemplo, siguiendo el convenio de C:

Objeto NodoEnteroALógico
...
Método códigoControl(cierto, falso)
r := e.generaCódigo()
si cierto≠ `None` **entonces**
emite(`bne` r, `$zero ,` cierto)
si no
emite(`beq` r, `$zero ,` falso)
fin si
si cierto≠`None` **y** falso≠`None` **entonces**
emite(`j` falso)
fin si
liberaReg(r);
fin códigoControl
...
fin NodoEnteroALógico

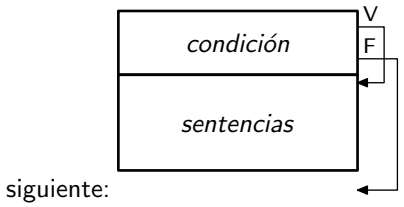
Las instrucciones `beq` y `bne` saltan si sus operandos son iguales o distintos, respectivamente.

4.2. Instrucciones condicionales

Para la instrucción condicional, tenemos dos versiones distintas según si existe o no la parte else. Si sólo tenemos la parte si,

si *condición* **entonces**
sentencias
fin si

tendremos que generar una estructura similar a esta:



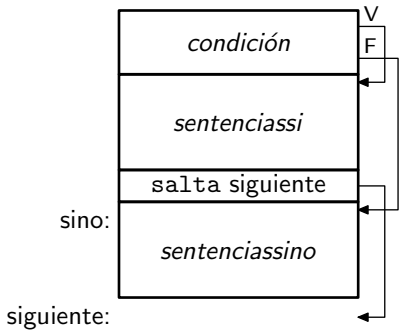
donde siguiente es una etiqueta que tendrá que generar el compilador. El bloque de condición saltará a esa etiqueta si la condición es falsa y a las instrucciones del bloque si es cierta. Podemos hacer esto en el método de generación de código de la siguiente manera:

Objeto NodoSi:
...
Método generaCódigo()
siguiente:=nuevaEtiqueta();
condición.códigoControl(`None`, siguiente);
para s **en** sentencias **hacer**
s.generaCódigo();
fin para
emite(siguiente`:`);
fin generaCódigo
...
fin NodoSi

Si tenemos la sentencia completa:

si *condición* **entonces**
sentenciassi
si no
sentenciassino
fin si

el esquema es el siguiente:



Ejercicio 15

Escribe el método de generación de código para la sentencia condicional completa.

Ejercicio 16

Escribe el resultado de generar código para la sentencia:

```
if x+3 && y then
  x:= x+2
else
  y:= y+1
fi
```

Supón que `x` e `y` tienen tipo entero y que están en las direcciones 1000 y 1004, respectivamente.

Ejercicio 17

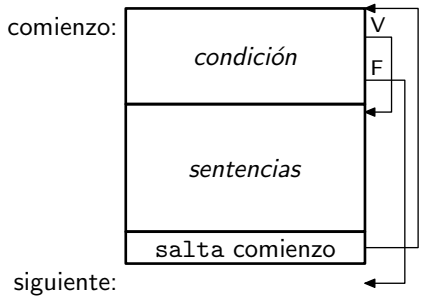
Escribe el método de generación de código para la sentencia condicional completa utilizando el método generaCódigo para la expresión. (Este es el que habría que utilizar si no tenemos control de flujo.)

4.3. Iteración

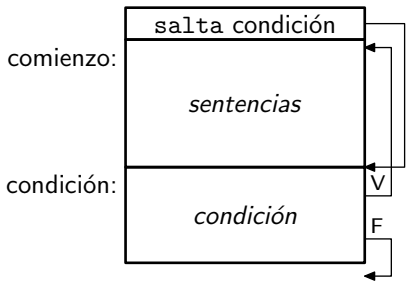
Veremos primero el bucle mientras. Si tenemos una estructura como:

mientras *condición* **hacer**
    sentencias
fin mientras

podemos pensar en un esquema similar a:



Sin embargo, puede ser más eficiente una estructura como la siguiente:



Ejercicio 18

¿Por qué es más eficiente el segundo esquema?
Pista: ¿cuántas veces se ejecuta el salto incondicional en cada caso?

Ejercicio 19

Escribe el esquema de generación de código para la sentencia mientras.

Ejercicio 20

¿Cómo podemos implementar instrucciones `break` y `continue` similares a las de C?

Vamos a ver ahora la compilación de la sentencia para según la sintaxis siguiente:

para *v*:=*expr1* **hasta** *expr2* **hacer**
    sentencias
fin para

Dado que sabemos compilar la sentencia mientras, es tentador transformarla de la siguiente manera:

v:=*expr1*;
mientras *v*≤*expr2* **hacer**
    sentencias
    v:=*v*+1;
fin mientras

Sin embargo, este esquema tiene un defecto importante: si *expr2* alcanza el máximo valor de su tipo, se producirá un desbordamiento al llegar al final. Para resolverlo, tenemos que cambiar el esquema:

v:=*expr1*;
si *v*> *expr2* **saltar** siguiente
bucle:
sentencias
si *v*= *expr2* **saltar** siguiente
v:=*v*+1;
saltar bucle
siguiente:

Observa que entre ambos esquemas hay una diferencia importante en cuanto al valor de la variable de control en la salida. Es habitual que los lenguajes de programación no establezcan restricciones acerca del valor de la variable, pero si lo hacen, podría ser necesario modificar el esquema para tenerlo en cuenta.

Finalmente, en muchos lenguajes de programación, la semántica del bucle para establece que las expresiones de los límites se evalúan sólo una vez. En ese caso, habría que crear una variable temporal y almacenar en ella el valor de *expr2*.

Ejercicio 21

Diseña la generación de código para una sentencia for similar a la de C. ¿Cómo se implementarían las instrucciones break y continue?

4.4. Selección múltiple

Terminaremos con la instrucción de selección múltiple. Esta estructura se puede implementar de varias maneras con distintos grados de eficiencia. La forma general de la estructura es:

opción *expresión*
    v_1: *acción*$_1$;
    v_2: *acción*$_2$;
    ...
    v_n: *acción*$_n$;
    otro: *acción*$_d$;
fin opción

La implementación más directa es transformarla en una serie de sentencias si:

t:=*expresión*;
si t=v_1 **entonces**
    acción$_1$;
si no si t=v_2 **entonces**
    acción$_2$;
...
si no si t=v_n **entonces**
    acción$_n$;
si no
    acción$_d$;

donde t es una variable temporal. Sin embargo, es ligeramente más eficiente algo parecido a:

t:=*expresión*;
si t=v_1 **saltar** et1;
si t=v_2 **saltar** et2;
...
si t=v_n **saltar** etn;
acción$_d$;
saltar siguiente;
et1: *acción*$_1$;
saltar siguiente;
et2: *acción*$_2$;
saltar siguiente;
...
etn: *acción*$_n$;
siguiente:

En cualquier caso, la búsqueda de la opción correcta tiene un coste lineal con el número de opciones. Esto es aceptable cuando hay pocas (digamos, diez o menos). Si el número de opciones es elevado, se puede crear una tabla de pares valor-dirección y generar código que busque en la tabla. En caso de que los valores posibles estén en un rango $i_1 \dots i_n$, y haya prácticamente $i_n - i_1$ valores distintos, se puede hacer que la tabla tenga la forma de un array indexado de i_1 a i_n con una dirección en cada celda. La sentencia case consistiría entonces en saltar a la dirección almacenada en la celda con índice igual al valor de la expresión. Si el número de valores es reducido con respecto al tamaño del rango, interesa generar código que implemente bien una búsqueda en una tabla de dispersión (tabla *hash*) o bien una búsqueda binaria.

5. Organización y gestión de la memoria

Antes de entrar en la generación de código para las funciones, comentaremos cómo organizaremos la memoria ya que esto tiene una repercusión directa en el modo en que se efectúan los saltos a las funciones y los retornos de ellas.

Una primera división de la memoria es la que distingue entre la memoria del programa y la destinada a los datos. Hay poco que decir sobre la memoria ocupada por el programa. Su tamaño es conocido en tiempo de compilación y, en la mayoría de los lenguajes modernos, se puede decir que es "invisible": el código no puede modificarse a sí mismo.

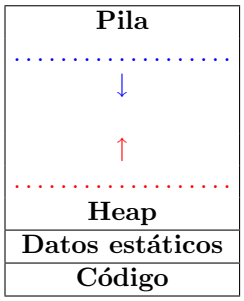
En cuanto a la memoria destinada a datos, podemos dividirla entre la memoria *estática*, que se gestiona en tiempo de compilación, y la *dinámica*, que se gestiona en tiempo de ejecución.

La memoria destinada a datos estáticos se utiliza para las variables globales y algunas otras como las variables estáticas de las funciones.

Al hablar de la memoria gestionada en tiempo de ejecución, podemos distinguir entre la memoria que se utiliza para albergar los objetos que se crean y destruyen con la ejecución de las funciones (parámetros, variables locales y algunos datos generados por el compilador) y la que se suele conocer como memoria dinámica, que se reserva explícitamente por el programador o que se necesita para almacenar objetos con tiempos de vida o tamaños desconocidos en tiempo de compilación. La memoria asociada a las funciones será gestionada mediante una pila. La memoria dinámica se localizará en un *heap*.

La memoria dinámica puede ser gestionada explícitamente por el programador (por ejemplo, mediante llamadas a `malloc` y `free`) o por el soporte en ejecución mediante técnicas de *garbage collection*. Nosotros sólo estudiaremos aquí la gestión de la memoria de pila.

Normalmente, la memoria se organiza de una manera similar a:



Es habitual que la pila crezca "hacia abajo" debido a que muchos procesadores tienen instrucciones que facilitan la creación de pilas "descendentes". Dado que la pila puede colisionar con el *heap*, el compilador deberá generar código para que en caso de colisión el programa pueda pedir memoria adicional al sistema o terminar adecuadamente. Normalmente, el sistema operativo ofrece ayuda tanto para la detección de la colisión (suele hacerlo con ayuda del hardware) como para facilitar el aumento de tamaño del bloque. Obviamente, la organización dependerá en última instancia del sistema operativo que es el que determina el modelo de proceso que se utiliza.

Sin embargo, en los restantes ejemplos supondremos que la pila crece hacia arriba ya que esta es la forma más cómoda de implementarla en la máquina virtual que utilizamos en prácticas.

5.1. Memoria estática

La gestión de la memoria estática es la más sencilla. De hecho, algunos lenguajes de programación antiguos, como las primeras versiones de FORTRAN, únicamente tienen reserva estática de memoria. Las principales ventajas son la sencillez de implementación y que los requerimientos de memoria del programa son conocidos una vez compilado el mismo.

Sin embargo, tiene bastantes inconvenientes dado que no permite trabajar con objetos de longitud variable. Por eso, es difícil para el programador definir el tamaño de las estructuras que va a usar: si son demasiado grandes, desperdiciará memoria; si son pequeñas, no podrá utilizar el

programa en todos los casos. Además, sólo puede haber una instancia de cada objeto por lo que no se pueden implementar procedimientos recursivos.

La gestión de esta memoria se puede hacer íntegramente en tiempo de compilación, basta con asignar a cada objeto una dirección. El compilador puede tener una variable, *dirLibre*, que inicializa con el valor de la primera dirección libre de memoria. A cada objeto global, se le asigna el valor actual de *dirLibre* y esta se incrementa en la talla del objeto.

El valor inicial de *dirLibre* dependerá de la arquitectura de la máquina que vayamos a utilizar. Si el espacio de direcciones se dispone como hemos comentado al comienzo del tema, el valor inicial deberá ser la dirección inmediatamente posterior al código. Dado que la longitud de éste no se conoce hasta el fin de la compilación, puede ser necesario generar las direcciones relativas a una etiqueta y dejar que el ensamblador o el enlazador las resuelva adecuadamente. Por otro lado, si la máquina permite que los datos estén en su propio segmento o se va a utilizar un enlazador que los pueda mover, lo habitual es comenzar por la dirección cero.

5.2. Memoria de pila

Supondremos que el programa se compone de una serie de funciones, entre las cuales hay una principal. Si la función no devuelve nigún valor, diremos que es un *procedimiento*. En cada llamada a intervienen dos funciones: el *llamador* (*caller*) y el *llamado* (*callée*). Los parámetros de la función en la declaración son los llamados *parámetros formales* y los valores que reciben en la llamada son los *parámetros de hecho* (*actual parameters*) o *argumentos*.

Cuando se llama a una función, se debe reservar espacio para guardar tanto los parámetros como las variables locales. Además, será necesario guardar cierta información adicional para volver al punto de llamada y para gestionar las variables. La región de memoria que asociamos a cada activación de la función se llama *registro o trama de activación*.

Dado que es posible que en un momento dado haya varias funciones activas simultáneamente (o varias activaciones simultáneas de la misma función si es recursiva), habrá que prever la posibilidad de tener diversos registros de activación. Nos concentraremos en el modelo más sencillo: las funciones se ejecutan desde el inicio de su cuerpo y no devuelven el control al punto en que fueron llamadas hasta finalizar su ejecución[3]. En este caso, podemos organizar los registros de activación en forma de pila.

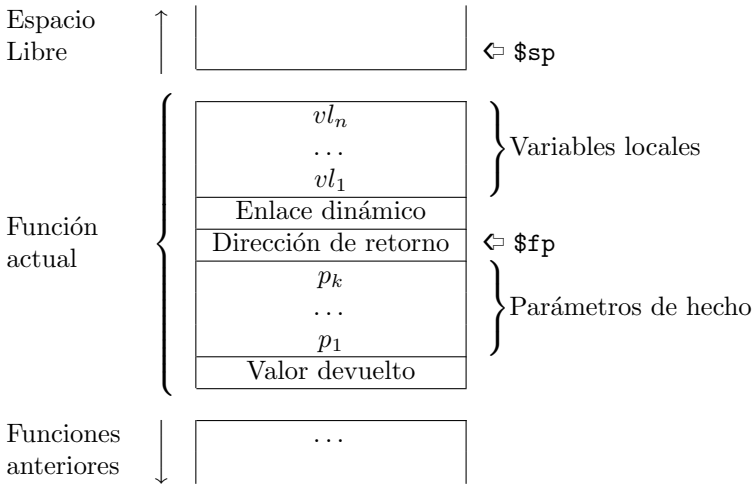
Los detalles de cómo se organizará la pila o cómo se hará la llamada dependerán de muchos factores. Entre otros, de qué instrucciones están disponibles, de lo que defina el sistema operativo o de la necesidad de que se pueda acceder a las funciones desde lenguajes distintos.

Supondremos que disponemos de dos registros especiales. Mantendremos un puntero a la primera posición libre de memoria tras la pila en el registro `$sp` (*Stack Pointer*). Para poder pasar parámetros y utilizar variables locales, utilizamos el registro `$fp` (de *Frame Pointer*, puntero de trama). La idea es acceder uniformemente a variables locales y parámetros calculando su dirección a partir del desplazamiento respecto de la dirección apuntada por `$fp`. Así, la tabla de símbolos trata de modo similar los parámetros y las variables locales: registra su posición en memoria con un *desplazamiento* sobre el registro `$fp`, y no con una dirección absoluta. De este modo, el registro de activación tendrá lo siguiente:

- espacio para el *valor devuelto* (opcional),
- los *parámetros de hecho*,
- la *dirección de retorno* de la función,
- el *enlace dinámico* (el `$fp` del llamador),
- las *variables locales*.

Esquemáticamente, lo podemos representar así:

[3] Es decir, no contemplamos la posibilidad de utilizar, por ejemplo, sentencias como la sentencia `yield` de Python.



Aquí seguimos el convenio de hacer que se acceda a las variables locales con desplazamientos positivos sobre `$fp`, y a los parámetros con desplazamientos negativos sobre `$fp`.

Cuando desactivamos un registro, `$fp` pasa a apuntar donde diga el *enlace dinámico*, que contiene el valor de `$fp` que teníamos antes de activar la función que ahora desactivamos. Este orden se aplica si la pila crece "hacia arriba", lógicamente, si la pila crece "hacia abajo", el orden se invierte. Observa que de esta manera, el llamador no necesita preocuparse de las variables locales de la función para colocar los parámetros, le basta con apilarlos. El llamado puede después preparar la pila a su gusto.

Por ejemplo, el registro de activación para

```
int f(int a, int b) {
  int c, d;
  ...
}
```

será de la forma

d
c
Enlace dinámico
Dirección de retorno
a
b
Valor devuelto

5.3. Funciones y tabla de símbolos

La tabla de símbolos debe registrar cierta información sobre cada función definida en el programa. En la declaración de la función f anotamos en la tabla de símbolos:

- el tipo del valor devuelto y su tamaño,
- el tipo de cada parámetro formal, su tamaño y su "distancia" a `$fp`,
- finalmente, anotamos también el tamaño total del registro de activación que corresponderá a f; este valor es la suma de los tamaños anteriores, del tamaño de otros campos del registro (la dirección de retorno, el enlace dinámico, etc.) y del tamaño total de la variables locales definidas en la función.

Aparte, para cada parámetro y variable local anotamos una entrada con su nombre y la información que necesitamos de ella (tipo, dirección, etc.).

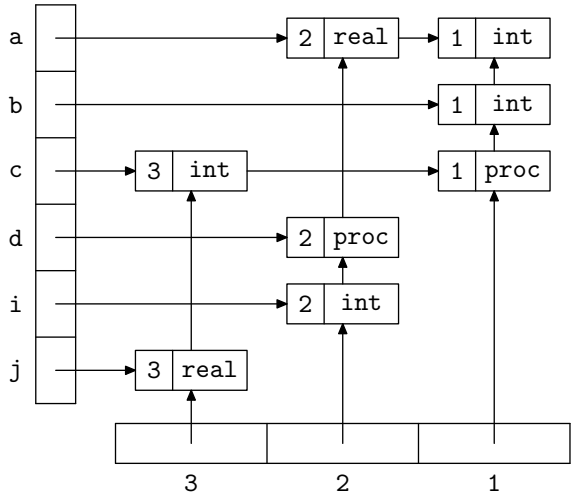
Además, debemos tener en cuenta que los identificadores declarados en el interior de una función dejan de estar disponibles al salir de ella. Por eso, es necesario que la tabla de símbolos sea

capaz de reflejar los ámbitos del programa. Esto sugiere añadir a las operaciones de la tabla dos más: *entraFunción* y *salFunción*. La primera la utilizaremos cuando entremos en una función para señalar que las nuevas declaraciones estarán en su interior él. La segunda es la que se encargará de eliminar las declaraciones contenidas en la función y que dejan de tener efecto.

La forma de funcionamiento de la tabla sugiere una organización de la información de los identificadores similar a una pila. Tendremos una tabla *hash* de manera que la información de cada identificador será un puntero a una pila. Al encontrar una declaración, la información adecuada se almacenará en el tope de la pila. Al salir de la función, se eliminarán todas las declaraciones creadas en su interior. Para facilitar esta eliminación, se puede utilizar una lista que enlace las entradas que se creen en la función.

Aquí podemos ver la situación de la tabla en un momento del análisis de un programa:

```
PROGRAM ejemplo;
    VAR
        a,b: integer;
    PROCEDURE c(i:integer);
        VAR
            a: real;
        PROCEDURE d(j:real);
            VAR
                c: integer;
        ...
```



6. Generación de código para las llamadas a función

Para llamar a las funciones utilizaremos la instrucción `jal`, que salta a una etiqueta y guarda en el registro `$ra` la dirección siguiente a esta instrucción.

Supongamos inicialmente que tenemos la función *f* que no tiene parámetros ni valor de retorno. En este caso, para llamar a *f*, con etiqueta `func_f`, basta con hacer

```
jal func_f
```

Al comienzo del cuerpo de *f*, tendremos que crear el registro de activación con las siguientes instrucciones:

```
func_f:
  sw $ra, 0($sp)   # Guardamos la dirección de retorno.
  addi $sp, $sp, 1 # Incrementamos $sp
```

Al finalizar *f*, se deben recuperar la dirección de retorno y eliminar el registro de activación:

```
  subi $sp, $sp, 1 # Restauramos $sp
  lw $ra, 0($sp)   # Recuperamos la dirección de retorno
  jr $ra           # Volvemos
```

En general, la llamada a la función tiene que hacer más trabajo por dos aspectos: debemos reservar espacio para los objetos de la función (parámetros y variables locales) y debemos decidir qué hacer con los registros que estén en uso en el momento de la llamada. Por ejemplo, al generar código para la expresión `f(a,b)*f(c,d)`, se guardará el resultado de `f(a,b)` en un registro. Dado que este registro puede ser utilizado durante la ejecución de `f(c,d)`, habrá que asegurarse de

que tenga el valor correcto a la hora de hacer la multiplicación. Tenemos varias posibilidades. El convenio *caller-saves* estipula que el llamador guarda cualquier registro que le interese conservar. Otro convenio es el *callee-saves*, con el que el llamado guarda aquellos registros que va a modificar. También se pueden mezclar ambos convenios, de esta manera algunos registros son responsabilidad del llamado y otros del llamador. Por simplicidad, supondremos que el llamador guarda en la pila los registros que quiera conservar.

Así pues, el código correspondiente a una llamada a la función f hace lo siguiente:

- Guarda los registros activos.
- Calcula el valor o dirección de cada parámetro de hecho y lo apila en el registro de activación.
- Apila la dirección de retorno y salta a la función.
- Recupera los registros activos.

El código de la función incluirá un prólogo que debe realizar las siguientes acciones:

- Rellenar la información del enlace dinámico en el registro de activación.
- Actualizar los registros `$sp` (sumándole el tamaño total de las variables locales, la dirección de retorno y el enlace dinámico) y `$fp`.

El código generado para una llamada a una función f con dos parámetros enteros a y b, sería:

```
...             # Guardar registros
...             # Cálculo de a (en $r1)
sw $r1, 0($sp)
add $sp, $sp, 1
...             # Cálculo de b (en $r2)
sw $r2, 0($sp)
add $sp, $sp, 1
jal rut_f       # Llamada a f
...             # Recuperar registros
```

Puedes pensar que podemos ahorrar alguna instrucción si agrupamos los incrementos del `$sp` al final o, mejor aún, si el incremento se realiza en el llamado. Es cierto, pero de hacerlo así se nos complica la generación de código por dos aspectos. El más sencillo de solucionar es que el almacenamiento en la pila ya no se hace con un desplazamiento 0 sino que hay que ir cambiándolo en cada parámetro. El problema más complejo es que puede suceder que tengamos llamadas anidadas (por ejemplo `f(f(1,2),3)`), lo que, si no se tiene mucho cuidado con el valor de `$sp`, puede hacer que el registro de activación de la llamada interna *machaque* el que estamos construyendo.

El prólogo de f sería:

```
rut_f:
  sw $ra, 0($sp)      # Guardamos $ra
  sw $fp, 1($sp)      # Guardamos $fp
  add $fp, $sp, $zero # Nuevo $fp
  addi $sp, $sp, 4    # Incrementamos $sp
```

Y el epílogo:

```
  subi $sp, $sp, 7 # Restauramos $sp
  lw $ra, 0($fp)   # Recuperamos $ra
  lw $fp, 1($fp)   # Restauramos $fp
  jr $ra           # Volvemos
```

Es interesante darse cuenta de la asimetría entre lo que sumamos a `$sp` en el prólogo y lo que restamos en el epílogo. Esto es debido a que podemos aprovechar el epílogo para *limpiar* los parámetros de hecho. También puedes darte cuenta de que esta asimetría es la que hace que usemos `$fp` en lugar de `$sp` para restaurar la dirección de retorno y el propio `$fp`.

7. Generación de código máquina

Esta fase recibe a la entrada un programa en un lenguaje intermedio (código de pila, de tres direcciones, estructurado como árbol, etc) y emite código de máquina para la máquina objetivo. La generación de código de máquina con el correspondiente optimizador forman el *back end* del compilador. Debe escribirse un *back end* para cada máquina para la que deseamos generar código.

Usualmente, el código de entrada ya ha sufrido una primera fase de optimización que permite eliminar cierta redundancia del código, aprovechar mejor el espacio (eliminando las variables temporales que no se necesitan), entre otras mejoras posibles.

7.1. Traducción de código intermedio a intrucciones de máquina

El proceso de traducción directo es relativamente sencillo. Cada instrucción del código intermedio puede traducirse por una secuencia de instrucciones de máquina.

Así las instrucciones:

```
lw $r1, -3($fp)
lw $r2, -2($fp)
add $r1, $r1, $r2
sw $r1, -1($fp)
```

pueden traducirse al ensamblador de Pentium por:

```
movl    -12(%ebp), %eax
addl    -8(%ebp), %edx
addl    %edx, %eax
movl    %eax, -4(%ebp)
```

Como ves, hemos tenido que emplear los registros disponibles en el procesador. Además, hemos ajustado los tamaños de modo que las variables que ocupaban una posición de memoria ahora ocupan cuatro.

7.2. Selección de instrucciones

Aunque seguir el proceso mencionado antes produciría código ejecutable correcto, si se quiere hacer que el código sea eficiente hay que tener cuidado con cómo se hace la selección. En particular, muchos lenguajes de máquina tienen instrucciones especializadas que permiten ahorrar espacio, accesos a memoria y/o una ejecución más eficiente.

Por ejemplo, si tenemos

```
lw $r1, -1($fp)
addi $r2, $zero, 1
add $r1, $r1, $r2
sw $r1, 0($fp)
```

es mejor emplear instrucciones de incremento:

```
leal -4(%ebp), %eax
incl (%eax)
```

Estas cuestiones están en la frontera entre generación de código y optimización de código.

7.3. Una pasada/múltiples pasadas

Los generadores de código de máquina pueden ser *de una pasada* o de *múltiples pasadas*, en función del número de veces que debe leerse el código intermedio. El principal problema lo plantean los *saltos hacia adelante*. Cada vez que encontramos una definición de etiqueta, recordamos la

dirección de memoria en la que se encuentra. Así, cuando encontramos un salto con posterioridad, sabremos sustituir la etiqueta por la dirección que le corresponde. Pero si encontramos un salto a una etiqueta que aún no está definida no podemos efectuar esa sustitución.

Hay dos formas básicas de resolver el problema de los saltos hacia adelante:

- *Backpatching*: (una sola pasada) se mantiene una tabla de instrucciones que referencien etiquetas no definidas previamente. Cada instrucción con una referencia hacia adelante genera una instrucción incompleta. Cada definición de etiqueta se almacena en una tabla de etiquetas con su dirección. Cuando hemos finalizado de leer el código de entrada, el generador de código puede visitar la tabla de instrucciones incompletas y completar los datos que faltan.
- En generadores de múltiples pasadas, la primera de ellas averigua dónde se define cada etiqueta. Una segunda pasada reemplaza todas las referencias a etiquetas por sus respectivos valores.

Otra manera de resolver el problema es generar código ensamblador y que un ensamblador resuelva el problema. Lógicamente, el ensamblador se encontrará con el mismo problema y usará alguna de las estrategias anteriores. Esta opción es muy interesante por sí misma, ya que este es sólo uno de los múltiples problemas que tiene el paso de una representación en lenguaje ensamblador a una representación en lenguaje de máquina.

7.4. Reserva de registros

Aunque cada máquina presenta una arquitectura en principio distinta, una característica común a casi todas es disponer de un juego de registros (unidades de memoria especiales mucho más rápidas que la memoria convencional y/o que permiten ejecutar operaciones que no pueden hacerse directamente con la memoria).

Por ejemplo, hay máquinas (PDP-8) con un solo registro aritmético (llamado acumulador) sobre el que se efectúan todas las operaciones aritméticas. Otras (80x86) tienen un reducido conjunto de registros que no son de propósito general, es decir, sobre cada uno puede realizarse un conjunto determinado de operaciones. (En estas máquinas no se reduce el problema de la reserva de registros: muchas de las operaciones deben hacerse en registros particulares.)

Las arquitecturas más modernas tienen un número moderado o grande de propósito general. Por ejemplo, M680x0 tiene 16 registros de propósito general y las arquitecturas RISC pueden tener más de 500 registros (divididos en bancos de, por ejemplo, 32 registros utilizables simultáneamente).

Existen varios problemas a la hora de decidir cómo se usan los registros. Si nuestro lenguaje intermedio opera sobre pila, habrá que transformar parte de las operaciones para que empleen los registros del procesador. Por otro lado, los lenguajes intermedios que utilizan registros suelen tenerlos en un número ilimitado y los manejan de manera que todos los registros tienen las mismas propiedades. Esto hace que la asignación de registros a registros reales o direcciones de memoria no sea trivial. Por otro lado, si el lenguaje intermedio opera sólo sobre direcciones de memoria, habrá que hacer que algunas variables puedan estar en registros, al menos en determinados momentos. De esta manera se incrementará la eficiencia del código generado.

Una manera de reducir la complejidad es seguir una política que determine qué variables deben estar en registros en qué períodos de tiempo. Esta política puede venir dada por el sistema operativo, la arquitectura o por el autor del compilador. Así, se pueden reservar algunos registros para variables intermedias, otros para paso de parámetros, etc. Estos conjuntos no tienen por qué ser disjuntos, pero debe estar definido qué papel juega cada registro en un momento dado. Otro aspecto importante es decidir si es el llamador o el llamado el encargado de guardar aquellos registros que puedan variar en una función.

Podríamos pensar que la reserva es *optimización* de código, pero (al menos parte) suele encontrarse en la fase de generación de código. Una correcta selección de registros reduce el número de instrucciones y reduce el número de referencias a memoria. Esto último es particularmente importante en máquinas RISC, en las que se procura ejecutar una instrucción por ciclo de reloj.

8. Resumen

- Dos fases en la generación de código:
  - Código intermedio.
  - Código de máquina (o ensamblador).
- Máquinas virtuales:
  - Fácil generar código para ellas.
  - Pocas restricciones (p.ej. número infinito de registros).
  - Pocas instrucciones.
- Código para expresiones:
  - Evaluamos operandos.
  - Calculamos y guardamos el resultado.
  - Coerción de tipos:
    - Implícita en el nodo del operador.
    - Con nodos explícitos.
- Acceso a vectores:
  - Desde una dirección base "virtual".
  - Se genera código para el índice y se suma a la dirección base.
- Expresiones lógicas:
  - Mediante valores explícitos.
  - Mediante control de flujo.
- Estructuras de control:
  - Interesante evaluar la condición mediante control de flujo.
  - La evaluación de los límites de la sentencia para puede ser delicada.
  - Distintas posibilidades para la selección múltiple.
- Tres tipos de memoria:
  - Código.
  - Memoria gestionada en tiempo de compilación: estática.
  - Memoria gestionada en tiempo de ejecución: pila y *heap*.
- Memoria estática:
  - Sencilla.
  - Limitada.
- Memoria de pila:
  - Para la información de las funciones.
  - Registros de activación: parámetros, variables locales, otros.
- Generación de código máquina:
  - Se traducen las instrucciones de código intermedio a instrucciones de máquina.
  - Es importante hacer bien la selección.
  - Puede necesitarse más de una pasada (o dejárselo a un ensamblador).
  - La reserva de registros es crítica.