Universidad de Huelva

Tema 7

Generación de código

Índice

La arquitectura front-end/back-end

- El objetivo final de un compilador es traducir el código fuente a código máquina (o ensamblador) de la plataforma de destino.
- Muchos compiladores presentan una arquitectura dividida en dos partes: *front-end* y *back-end*.
  - El *front-end* realiza una traducción de código fuente a código intermedio.
  - El *back-end* realiza una traducción de código intermedio a código máquina.
- Esta división permite generar familias de compiladores que comparten el *front-end*.

Características del código intermedio

- Sus instrucciones corresponden básicamente a las instrucciones generales de los procesadores.
- Es independiente de la máquina (hasta cierto punto).
- No tiene en cuenta características como la arquitectura del procesador, el número de registros del procesador o la funcionalidad de estos registros.
- Aunque se pueda expresar en modo texto, el código intermedio se almacena en memoria en forma de árbol (ASA).
- El modo texto se utiliza para depurar el compilador.

Otras opciones

- Lenguajes pseudointerpretados (Java). El compilador genera un código binario de una máquina virtual. Este código es interpretado en tiempo de ejecución.
- Traductores fuente-fuente. La idea es traducir un código fuente a un lenguaje de alto nivel diferente (generalmente C) para el que ya exista un compilador eficiente. Esta solución se aplica comúnmente para lenguajes declarativos (Prolog, Haskell) o para producir código para microprocesadores que se distribuyan con un compilador de C.

Características generales

- Se suele tomar como código intermedio.
- Está formado por instrucciones con un máximo de tres direcciones: operando1, operando2 y resultado.
- Las instrucciones se representan por cuartetos: (operador, operando1, operando2, resultado).
- No todas las direcciones deben estar ocupadas. Existen instrucciones con menos direcciones.
- Las instrucciones son muy parecidas a las de cualquier ensamblador: sumas, restas, multiplicaciones, saltos condicionales, saltos incondicionales, ...

Características generales

- Las direcciones de memoria se gestionan de manera simbólica, es decir, no se asigna un valor numérico a estas direcciones.
- No se trata de ningún estándar. Cada familia de compiladores define su propia representación intermedia.
- Al diseñar un compilador hay total libertad a la hora de definir las instrucciones del código intermedio.
- Por ejemplo, la familia de compiladores *gcc* utiliza el código intermedio RTL (Register Transfer Language). Los front-end de C y C++ definen además otras representaciones previas llamadas GENERIC y GIMPLE que facilitan la optimización.

Instrucciones típicas

- Instrucciones básicas:
  - x := y op z (operación binaria aritmética o lógica)
  - x := op y (operación unaria aritmética o lógica)
  - x := y (asignación simple)
  - goto etiqueta (salto incondicional)
  - if x oprelacional y goto etiqueta (salto condicional)
- Instrucciones de asignación con índice
  - x[i] := y
  - x := y[i]

Instrucciones típicas

- Instrucciones de asignación con punteros
  - x := &y
  - x := *y
  - *x := y
- Instrucciones de llamadas a funciones:
  - param x (almacena el parámetro en la pila)
  - pop x (saca el valor de la pila y lo almacena en x)
  - call etiqueta, n (llama a la función que comienza en la etiqueta tomando n parámetros de la pila)
  - return y (devuelve el valor y)

Características generales

- Las instrucciones que aparecen en el código fuente suelen ser las siguientes:
  - Declaraciones de variables
  - Asignaciones
  - Bucles (while, do-while, for)
  - Instrucciones condicionales (if-then, if-then-else, switch-case)
  - Saltos (return, break, continue)
  - Bloques de instrucciones
- Estas instrucciones hacen uso de las siguientes construcciones:
  - Expresiones aritméticas
  - Condiciones
  - Llamadas a funciones

Declaraciones

- Las declaraciones de variables no generan código.
- La declaración de una variable debe crear una entrada en la tabla de símbolos.
- Esta información incluye el identificador de la variable, el tipo de dato (que permite calcular su tamaño) y su posición de memoria (desplazamiento respecto a una posición conocida)
- El compilador debe mantener un contador para asignar el desplazamiento de forma consecutiva.
- Las variables declaradas en distintos niveles de anidamiento se tratan como variables locales normales a la hora de asignarles memoria. El analizador semántico debe garantizar que no se acceda a estas variables en un ámbito diferente.

Expresiones

- La sintaxis de las expresiones en el lenguaje fuente es de la siguiente forma:
  - Expresión → Expresión **sum** Término | Término
  - Término → Término **prod** Factor | Factor
  - Factor → **literal** | **id** | **lpar** Expresión **rpar**
- Las expresiones se almacenan en forma de árbol dentro del ASA.
- El código asociado a una expresión está formado por una lista de instrucciones en código intermedio que permite evaluar la expresión.
- El resultado de la expresión se almacena en una variable (*temp*).

Expresiones

- Para generar el código asociado a la expresión se utiliza una función que genere variables temporales (*getNewTemp*()).
- Código asociado a una constante:
  - (A) Lista de instrucciones vacía. La variable *temp* es el valor de la constante.
  - (B) Instrucción de asignación de la constante a una nueva variable temporal.
- Código asociado a una variable:
  - Lista de instrucciones vacía. La variable *temp* es la variable referenciada.
- Código asociado a una operación unaria:
  - Se genera el código del operando
  - Se crea una nueva variable temporal *temp*.
  - Se añade la instrucción de asignación "*temp = operador operando.temp*".

Código de operando
temp = op temp1

(temp1)

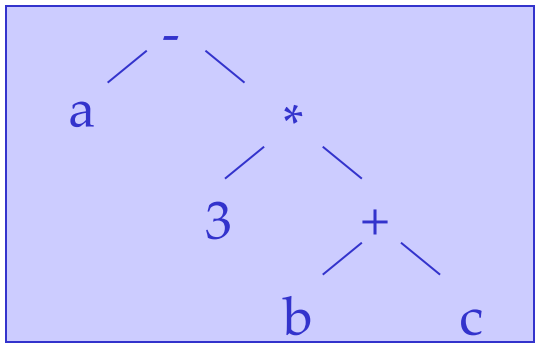
Expresiones

- Código asociado a una operación binaria:
  - Se genera el código de los operandos
  - Se crea una nueva variable temporal *temp*.
  - Se añade la instrucción de asignación

    "temp = operando1.temp op operando2.temp"

Código de operando1	(temp1)
Código de operando2	(temp2)
temp = temp1 op temp2	

- Ejemplo: a – 3 * (b + c)




```
tmp0 = 3
tmp1 = b + c
tmp2 = tmp0 * tmp1
tmp3 = a - tmp2
```

Condiciones

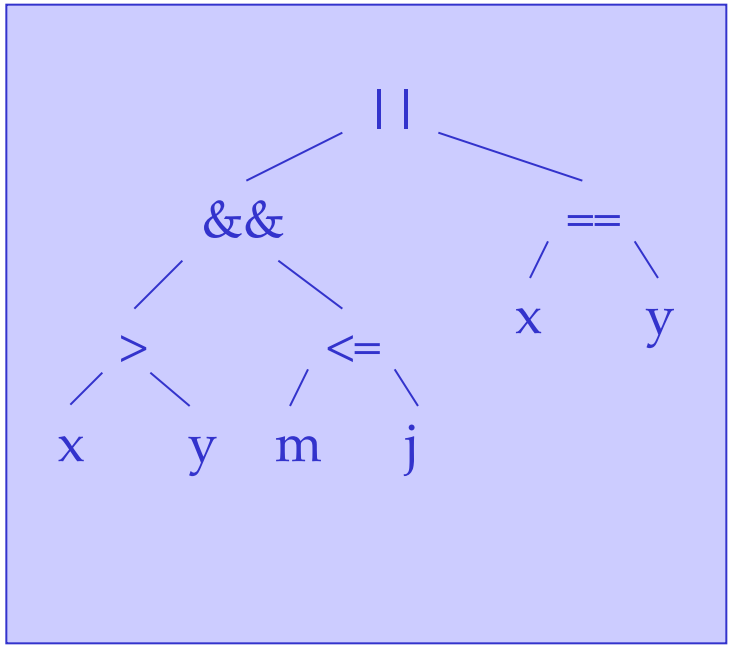
- Son expresiones booleanas que aparecen en los bucles e instrucciones condicionales.
- El código asociado debe generar un salto a dos posibles etiquetas:
  - *label_true*, para el caso verdadero
  - *label_false*, para el caso falso
- El código generado no escribe las etiquetas, sino que contiene saltos (condicionales o incondicionales) a esas etiquetas.
- Las condiciones se pueden representar de forma aritmética o por medio de saltos.

Condiciones

- Representación aritmética:
  - Números: (falso = 0, verdadero !=0), (falso<=0, verdadero>0)
  - Las condiciones se transforman en expresiones aritméticas:
    - c = a & b ; c=0 si a=0 ó b=0, c=1 si a!= 0 y b!=0
    - c = a | b ; c=0 si a=0 y b=0, c=1 si a!=0 ó b!=0
    - c = x > y ; c=0 si x<=y, c=1 si x>y
  - El código se genera de forma idéntica al de las expresiones.
  - Al final del código de la expresión se añaden los saltos a las etiquetas.

Condiciones

- Ejemplo: (x>y && m<=j) || x==y




```
tmp0 =  x>y
tmp1 = m<=j
tmp2 = tmp0 & tmp1
tmp3 = x==y
tmp4 = tmp2 | tmp3
if tmp4 != 0 goto label_true
goto label_false
```

Condiciones

- Representación con saltos (en cortocircuito)
  - Las expresiones booleanas se basan en saltos condicionales
  - Permite no evaluar la parte innecesaria de la condición (cortocircuito)
  - Comparaciones aritméticas (c = x > y)

```
if x>y goto label_true
goto label_false
```

  - Operaciones lógicas (c = a & b)

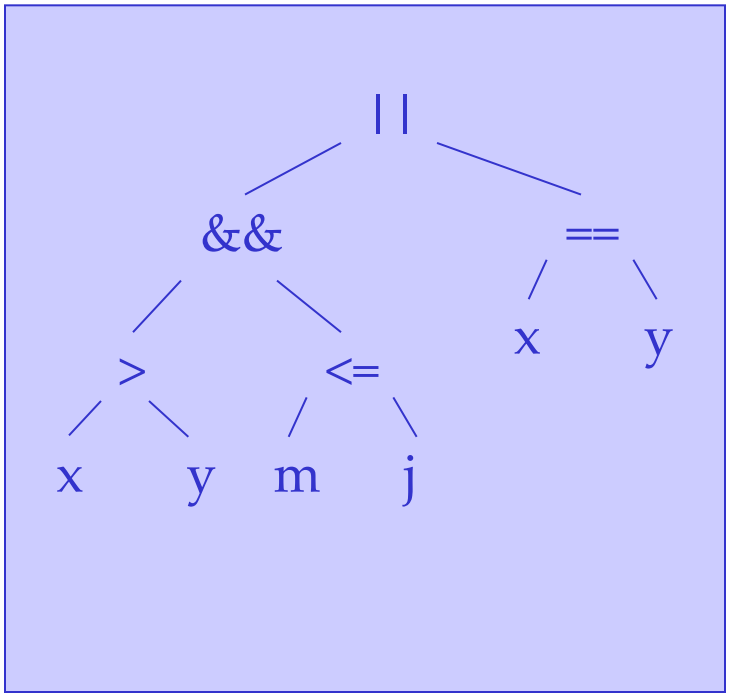
```
if a!=0 goto Etiqueta0
goto label_false
Etiqueta0:
if b!=0 goto label_true
goto label_false
```

(c = a | b)

```
if a!=0 goto label_true
goto Etiqueta0
Etiqueta0:
if b!=0 goto label_true
goto label_false
```

Condiciones

- Ejemplo: (x>y && m<=j) || x==y




```
    if x>y goto Etiqueta0
    goto Etiqueta1
Etiqueta0:
    if m<=j goto label_true
    goto Etiqueta1
Etiqueta1:
    if x==y goto label_true
    goto label_false
```

Asignaciones

- La sintaxis de la instrucción de asignación en el lenguaje fuente es:
  - InstAsig → **id** **asig** Expresión **pyc**
- Código asociado a una instrucción de asignación:
  - generando el código de la expresión
  - generar una instrucción "*id = Expr.temp*"

Código de la expresión	(temp)
id = temp	

Instrucción if-then

- La sintaxis de la instrucción *if-then* en el lenguaje fuente es:
  - InstIf → **if** Condición **then** Instrucción
- La estructura del código generado es

Código de la condición	label_true label_false
Cond.label_true:	
Código de la instrucción	
Cond.label_false:	

Instrucción if-then-else

- La sintaxis de la instrucción *if-then-else* en el lenguaje fuente es:
  - InstIfElse → **if** Condición **then** Instrucción1 **else** Instrucción2
- La estructura del código generado es

Código de la condición	label_true label_false
Cond.label_true:	
Código de la instrucción1	
goto label_end	
Cond.label_false:	
Código de la instrucción2	
label_end:	

Instrucción switch-case

- La sintaxis de la instrucción *switch-case* en el lenguaje fuente es:
  - InstSwitch → **switch** **lparen** Expresión **rparen** BloqueSwitch
  - BloqueSwitch → **lbrace** (SentenciaCase)* [SentenciaDefault] **rbrace**
  - SentenciaCase → **case** Valor **colon** (Instrucción)*
  - SentenciaDefault → **default** **colon** (Instrucción)*

Instrucción switch-case

- La estructura del código generado es

Código de la expresión	temp
if temp==valor1 goto label_case1	
if temp==valor2 goto label_case2	
goto label_default (o label_end)	
label_case1:	
Código del bloque case1	
label_case2:	
Código del bloque case2	
label_default:	
Código del bloque default	
label_end:	

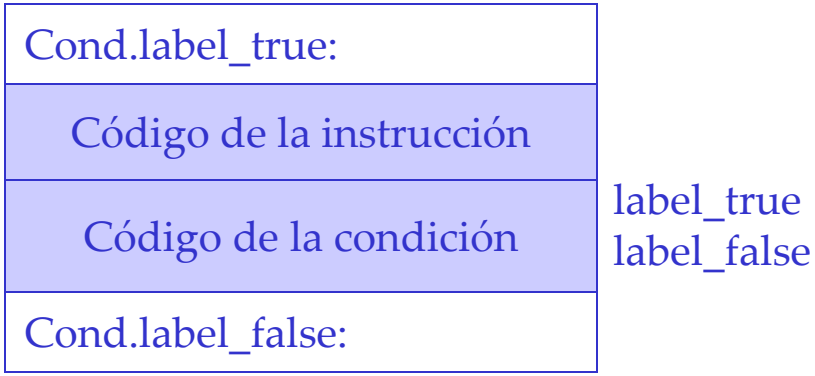
Instrucción while

- La sintaxis de la instrucción *while* en el lenguaje fuente es:
  - InstWhile → **while** Condición **do** Instrucción
- La estructura del código generado es

label_begin:	
Código de la condición	label_true label_false
Cond.label_true:	
Código de la instrucción	
goto label_begin	
Cond.label_false:	

Instrucción do-while

- La sintaxis de la instrucción de *do-while* en el lenguaje fuente es:
  - InstDoWhile → **do** Instrucción **while** Condición
- La estructura del código generado es



Instrucción for

- La sintaxis de la instrucción *for* en el lenguaje fuente es:
  - InstFor → **for** **lparen** Instrucción1 **semicolon** Condición **semicolon** Instrucción2 **rparen** Instrucción3
- La estructura del código generado es

Código de la instrucción1	
label_begin:	
Código de la condición	label_true label_false
Cond.label_true:	
Código de la instrucción3	
Código de la instrucción2	
goto label_begin	
Cond.label_false:	

Instrucción break

- Representa un salto incondicional (*goto*) hacia la etiqueta final de una instrucción *switch*, *while*, *do-while* o *for*.
- Requiere una pila de etiquetas, de manera que al entrar en una de estas instrucciones, se almacena la etiqueta final en la pila. Al salir de las instrucciones se desapila la etiqueta.

Instrucción continue

- Representa un salto incondicional (*goto*) hacia la etiqueta inicial de una instrucción *while*, *do-while* o *for*.
- Requiere una pila de etiquetas, de manera que al entrar en una de estas instrucciones, se almacena la etiqueta inicial en la pila. Al salir de las instrucciones se desapila la etiqueta.

Llamadas a funciones

- La sintaxis de la llamada a una función en el lenguaje fuente es:
  - InstCall → **identificador** **lparen** ListaParametros **rparen**
  - ListaParámetros → [Expresión (**comma** Expresión)*]
- La estructura del código generado es

Código Expresión1	temp1
param temp1	
Código Expresión2	temp2
param temp2	
.....	
call id.comienzo	