Tema 8. Generación de Representaciones Intermedias

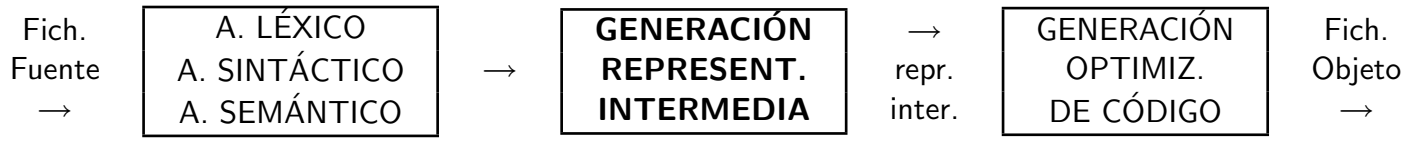
Francisco José Ribadas Pena

PROCESADORES DE LENGUAJES
4º Informática

ribadas@uvigo.es

7 de mayo de 2006

8.1 Introducción



Representación Intermedia (RI): Lenguaje de una máquina abstracta usado como interfaz entre Análisis y Gen. de Código.

- Etapa intermedia, previa a la generación de código objeto.
- Permite especificar operaciones sobre la máquina objetivo sin entrar en detalles de bajo nivel específicos de la arquitectura.
- Aisla el análisis (léxico, sintáctico, semántico) de la generación de código objeto.
- Muy relacionado con el desarrollo de intérpretes

Ventajas:

1. Mayor modularidad. (Ofrece abstracción para componentes de alto nivel)
   - Aisla elementos de más alto nivel de los dependientes de la máquina
2. Facilita optimización y generación de código
   - Elimina/simplifica características específicas máquina objetivo
     - nº limitado de registros, tipos de instrucciones
     - alineación de datos, modos de direccionamiento, etc,...
   - Permite optimizaciones independientes de la máquina
   - Ofrece representación simple y uniforme $\Rightarrow$ fácil generar/optimizar código
3. Mayor transportabilidad
   - Independencia de la máquina objetivo y del lenguaje fuente
     - Análisis no depende de arquitectura destino
     - Generación código no depende del leng. original
   - Crear un compilador para otra máquina $\Rightarrow$ basta crear una nueva etapa final de generación
   - Ejemplo: multicompiladores

Inconvenientes:

1. Necesidad de una fase extra para traducir a código máquina
   - Mayor coste computacional
2. Dificultad para definir un leng. intermedio adecuado
   - Compromiso entre la representación de elementos código fuente y del código máquina.

8.2 Tipos de Representaciones Intermedias

Características deseables de una R.I.:

- Sencillo de producir durante fase de A. Semántico
- Facilitar traducción al leng. máquina final para todas las posibles máquinas objetivo
- Construcciones claras, sencillas y uniformes, con significado unívoco
  - Facilita la especificación de la traducción a la cada máquina objetivo

Tipos de R.I.:

- Representaciones arbóreas: árboles de sintaxis abstracta, GDAs
- Representaciones lineales: polaca inversa, código 3 direcciones

Ejemplos de R.I:

Descriptive Intermediate Attributed Notation for Ada

- Diana: represent. arbórea usada en compiladores de ADA
- RTL (*register transfer language*): usada en familia de compiladores GCC
- Código-P: compiladores Pascal (también común en intérpretes)
- WAM (*Warren abstract machinne*): intérpretes Prolog
- Bytecodes JAVA

Ejemplo GCC.

8.2.1 R.I. Arbóreas

Basadas en árboles de sintaxis abstracta,

- Árboles de análisis sintáctico son info. superflua
- Nodos hoja $\longrightarrow$ operandos
- Nodos internos$\longrightarrow$ operadores del Lenguaje Intermedio

Ejemplos

```
s := a + b * c;
```

```
IF (a <5) THEN c := b;
```

```
WHILE (a >3) DO
   b := b + a;
   a := a + 1;
END DO;
```

Pueden usarse representaciones de más bajo nivel.

```
IF (a < 5) THEN c := b + 1;
```

También se usan GDAs (grafos dirigidos acíclicos)

- Representaciones condensadas de árboles abstractos
  - Subestructuras idénticas representadas una única vez
  - Uso de múltiples referencias a regiones comunes
- Mejora de tamaño y optimización de código implícita
- Ejemplo: `a := b * (-c) + b * (-c)`

8.2.2 R.I. Lineales

(a) Notación Polaca Inversa (RPN)

Notación postfija con los operadores situados a continuación de los operandos.

Ejemplo: `d := (a + b) * c` $\longrightarrow$ `d a b + c * :=`

Muy usado como código intermedio en intérpretes.

- Interpretación muy sencilla (basta una pila)
- Ejemplo: Postscript

Ventajas:

- Muy sencillo para expresiones aritméticas.
  - No necesita paréntesis
  - Precedencia está implícita $\longrightarrow$ valores temporales implícitos
- Interpretación/Generación de código muy simple
  - Sólo necesita una pila. Algoritmo interpretación:
    1. Recorrer lista, apilando operandos hasta llegar a un operador
    2. Tomar los operandos necesarios de la pila y aplicarles el operador
    3. Apilar resultado y continuar
  - Muy usado en primeras calculadoras comerciales

Inconvenientes:

- Difícil de entender (una sóla línea de código intermedio)
- Complicado representar control flujo (saltos)
- No es útil para optimización de código.

(b) Código de 3 Direcciones

Generalización del código ensamblador de una máquina virtual de 3 direcciones.

- Cada instrucción consiste en un operador y hasta 3 direcciones (2 operandos, 1 resultado)
- FORMATO GENERAL : `x := y OP z`
- `x, y, z` sin direcciones. Referencias a:
  - Nombres (dir. de variables)
  - Constantes
  - Variables temporales (creadas por el compilador durante la gen. de la R. I.)
  - Etiquetas (direcciones de instrucciones)
- Ejemplos:

```
d := x + 9 * z

temp1 = 9 * z
temp2 = x + temp1
d = temp2
```

```
IF (a<b) THEN x:=9

  temp1 = a<b
  if temp1=true goto etq1
  goto etq2
etq1:  x := 9
etq2:  ...
```

Existe una referencia explícita a los resultados intermedios de las operaciones mediante las vars. temporales.

$\longrightarrow$ En RPN, referencias implícitas a resultados almacenados en pila

Posibilidades de representación: { cuartetos, tercetos, tercetos indirectos

(b.1) CUARTETOS

Tuplas de 4 elementos.

(<operador>, <operando1>, <operando2>, <resultado>)

Ejemplos:

```
d := x + 9 * z          IF (a<b) THEN x:=9

(*,9,z,temp1)           (<,a,b,temp1)
(+,x,temp1,temp2)       (if_true,temp1, ,etq1)
(:=,temp2, ,d)          (goto, , ,etq2)
                        (label, , ,etq1)
                        (:=,9, ,x)
                        (label, , ,etq2)
```

(b.1) TERCETOS

Tuplas de 3 elementos.

(<operador>, <operando1>, <operando2>)

La dir. destino del resultado esta asociada de forma implícita a cada terceto.

- Existe una var. temporal asociada a cada terceto donde se guarda su resultado
- Valores intermedios se referencian indicando el n° del terceto que lo crea

Ejemplos:

```
d := x + 9 * z          IF (a<b) THEN x:=9

101:(*,9,z)             111:(<,a,b)
102:(+,x,[101])         112:(if_true,[111],etq1)
103:(:=,d,[102])        113:(goto,etq2, )
                        114:(label,etq1, )
                        115:(:=,x,9)
                        116:(label,etq2, )
```

Características:

- Más concisos que cuartetos (menos espacio)
- Evita manejo explícito de vars. temporales
- Complica optimización: mover/copiar/borrar tercetos más complejo
- Equivalen a arboles de sintaxis abstracta (con ramificación limitada a 2 descendientes)

```
s := a + b * c

1:(*,b,c)
2:(+,a,[1])
3:(:=,s,[2])
```

(b.1) TERCETOS INDIRECTOS

- Orden de ejecución de los tercetos determinada por un vector de apuntadores a triples (VECTOR de SECUENCIA)
- Referencias a vars. temporales se hacen directamente al terceto, no al vector de secuencia
- Representación más compacta $\longrightarrow$ no repetición de tercetos
- Indirección facilita mover/copiar/borrar $\longrightarrow$ simplifica optimiz.
- Ejemplo:

COD. 3 DIR.	TERCETOS	VECTOR SECUENCIA
`a := c / d`	101:(/,c,d)	1: **101**
`c := c + 1`	102:(:=,a,[101])	2: 102
`b := c / d`	103:(+,c,1)	3: 103
`m := a - b`	104:(:=,c,[103])	4: 104
	105:(:=,b,[101])	5: **101** (el valor de c ha cambiado)
	106:(-,[102],[105])	6: 105
	107:(:=,m,[106])	7: 106
		8: 107

8.3 Generación de Código Intermedio

Ejemplo de generación de C.I. para los constructores típicos de los lenguajes de programación.

(1) Lenguaje Intermedio:

- Instrucción de asignación: "**a := b**" $\rightsquigarrow$ **(:=,b, ,a)**
- Operadores binarios: "**x := a OP b**" $\rightsquigarrow$ **(OP,a,b,x)**
  - Op. aritméticos

Reales		Enteros	
$+_R$	ADDF	$+_E$	ADDI
$-_R$	SUBF	$-_E$	SUBI
$*_R$	MULF	$*_E$	MULI
$/_R$	DIVF	$/_E$	DIVI

  - Op. booleanos

$\&\&$	AND
$\|\|$	OR
$\oplus$	XOR

  - Op. relacionales

$>$	GT	$\leq$	LE
$<$	LT	$\geq$	GE
$=$	EQ	$\neq$	NE

- Operadores unarios: "**x := OP a**" $\rightsquigarrow$ **(OP,a, ,x)**
  - Cambio signo: $-$ MINUS
  - Conversión tipos
    - CONVF (convierte int a float)
    - CONVI (convierte float a int)
  - Lógicos

$\sim$	NOT
$\gg$	SHFR
$\ll$	SHFL

- Saltos
  - Incondicional: "**goto etq**" $\rightsquigarrow$ **(goto,etq,)**
  - Condicional: "**if x goto etq**" $\rightsquigarrow$ **(if,x,etq,)**
    Si x es $true$ salta a etq.
    Además, saltos con ops. relacionales ($>$, $<$, $==$, etc..)
    "**if x $>$ y goto etq**" $\rightsquigarrow$ **(if_GT,x,y,etq)**
  - Declaración etiquetas: "**etq: <...>**" $\rightsquigarrow$ **(label,etq, ,)**
    Etiqueta "etq" estará asociada a la siguiente instrucción.

- Llamadas a procedimiento
  Llamada al procedimiento `proc(x1, x2, ..., xN,` con n argumentos.

```
PARAM x1
PARAM x2
...
PARAM xN
call proc, N
```

  Fin llamada a procedim.: "**return valor**" $\rightsquigarrow$ **(return, valor, ,)**

- Acceso a memoria
  - Acceso indirecto:

    "**x := y[i]**" $\rightsquigarrow$ **(:=[],y,i,x)** (acceso)
    "**x[i] := y**" $\rightsquigarrow$ **([]:=,y,i,x)** (asignación)

    "`a[j]`" $=$ posición de memoria j unidades (bytes/palabras) después de la dirección de a.

```
x[10] := a+b
```

```
temp100 := a+b
x[10] := temp100
```

  - Punteros:

    "**x := &y**" (asigna a x la dirección de y)
    "**x := *y**" (asigna a x el contenido apuntado por la dir. guardada en y)
    "***x := y**" (guarda en la dir. guardada en la var. x el valor de y)

(2) Suposiciones:

- Mantendremos el cod. generado en un atributo asociado a los símbolos de la gramática
- 1 TDS por bloque + 1 TDS asociada a cada tipo registro
- Contenido entradas TDS
  - Información de direccionamiento: posición relativa (*offset*) respecto al inicio del área de datos del bloque de código actual
    $\longrightarrow$ variables locales se sitúan consecutivamente según orden de declaración
  - Tamaño de las variables:

char: 1	real: 8	record: $\sum$ tamaño campos
int: 4	puntero:4	array : tamaño elementos $\times$ num. elementos

- Etiquetas: asociadas a una instrucción del lenguaje intermedio
  $\longrightarrow$ se refieren a direcciones de memoria de la zona de instrucciones
- Temporales: dir. de memoria destinadas al almacenamiento de valores intermedios
  $\longrightarrow$ al generar cod. objeto se decidirá si se refieren a registro o una pos. de memoria

(3) Atributos

- **dir:** referencia a la dirección en memoria/registro asociada a un identificador(variable) o a un temporal
  $\longrightarrow$ todo no terminal de la gramática usado en expresiones tendrá asociada siempre una var. temporal
- **código:** guarda el C.I.(lista de cuartetos o tercetos) generado para una construcción del lenguaje

Funciones Se supondrá que están definidas las dos funciones siguientes

- **crearTemporal()**: genera una nueva variable temporal
- **generarCI(instruccion)**: genera la representación en CI de una instrucción de 3 direcciones
- **nuevaEtiqueta()**: genera una nueva etiqueta

8.3.1 CI para expresiones aritméticas

$S \longrightarrow$ **id** $:= E$	{ buscarTDS(id.texto) /* obtenemos: id.tipo, id.dir, id.tamaño */ S.codigo = E.codigo + generarCI('id.dir = E.dir') }
$E \longrightarrow E$ **op** E (**op**:+,-,*,/,mod,...)	{ E0.dir = crearTemporal() E0.codigo = E1.codigo + E2.codigo + generarCI('E0.dir = E1.dir OP E2.dir') }
$E \longrightarrow$ **op** E (**op**: -,∧,...)	{ E0.dir = crearTemporal() E0.codigo = E1.codigo + generarCI('E0.dir = OP E1.dir') }
$E \longrightarrow (E)$	{ E0.dir = E1.dir E0.codigo = E1.codigo }
$E \longrightarrow$ **const**	{ E.dir = crearTemporal() E.codigo = generarCI('E.dir = const') }
$E \longrightarrow$ **id**	{ buscarTDS(id.texto) /* obtenemos: id.tipo, id.dir, id.tamaño */ E.dir = id.dir E.codigo = <> }

8.3.1 (cont.) Conversión de Tipos

- Arrastrar y consultar tipo asociado a construcciones (tema anterior)
- Incluir código para conversión implícita cuando sea necesario
- Aplicar la instrucción de C.I. que corresponda a cada tipo de dato

Ejemplo: Operador sobrecargado $+$

$E \rightarrow E + E$

```
{ E0.dir = crearTemporal()
  E0.codigo = E1.codigo +
              E2.codigo
  if (E1.tipo = REAL) and (E2.tipo = INTEGER)
          E0.tipo = REAL
          E0.codigo = E0.codigo +
                      generarCI('E2.dir = CONV_F E2.dir')+
                      generarCI('E0.dir = E1.dir +R E2.dir')
  else if (E1.tipo = INTEGER) and (E2.tipo = REAL)
          E0.tipo = REAL
          E0.codigo = E0.codigo +
                      generarCI('E1.dir = CONV_F E1.pos')+
                      generarCI('E0.dir = E1.dir +R E2.dir')
  else if (E1.tipo = REAL) and (E2.tipo = REAL)
          E0.tipo = REAL
          E0.codigo = E0.codigo +
                      generarCI('E0.dir = E1.dir +R E2.dir')
  else if (E1.tipo = INTEGER) and (E2.tipo = INTEGER)
          E0.tipo = INTEGER
          E0.codigo = E0.codigo +
                      generarCI('E0.dir = E1.dir +I E2.dir')
  else
          E0.tipo = ERROR
          E0.codigo = <>
}
```

8.3.1 (cont.) Casos Especiales

(1) REGISTROS

Suponemos campos almacenados de forma consecutiva.

Acceso a campos

$E \longrightarrow$ **id**.**id**	`{ buscarTDS(id1.texto)`
	`/* obtenemos: tipo, dir. inicio, TDS registro (R) */`
	`R.buscarTDS(id2.texto)`
	`/* obtenemos: tipo, dir. relativa en registro */`
	`E.dir = id1.dir + id2.dir`
	`/* no es necesario generar código */`
	`}`

Escritura de campos

$S \longrightarrow$ **id**.**id** $:= E$	`{ buscarTDS(id1.texto)`
	`/* obtenemos info. de registro R */`
	`R.buscarTDS(id2.texto)`
	`S.codigo = E.codigo +`
	`generarCI('id1.dir[id2.dir] = E.dir')`
	`}`

(2) ARRAYS

Suponemos arrays de 1 dimensión de tipo *array(I,T)*

- T: tipo de los elementos, guardamos su tamaño en TDS
- I: rango del índice, guardamos valores limite en TDS

Acceso

$E \rightarrow \textbf{id}\ [E]$

```
{ buscarTDS(id.texto)
  /* obtenemos: tipo 'array(I,T)', dir inicio,
   tamaño tipo base T, limite inferior de I */
  E0.dir = crearTemporal()
  despl = crearTemporal() /* despl. en array */
  indice = crearTemporal() /* despl. en memoria */
  E0.codigo = E1.codigo +
      generarCI('despl = E1.dir - I.lim_inferior')+
      generarCI('indice = despl * T.tamaño')+
      generarCI('E0.dir = id.dir[indice]')
}
```

Escritura

$S \rightarrow \textbf{id}[E] := E$

```
{ buscarTDS(id.texto)
  /* obtenemos info. del array */
  S.dir = crearTemporal()
  despl = crearTemporal() /* despl. en array */
  indice = crearTemporal() /* despl. en memoria */
  S0.codigo = E2.codigo +
      E1.codigo +
      generarCI('despl = E1.dir - I.lim_inferior')+
      generarCI('indice = despl * T.tamaño')+
      generarCI('id.dir[indice] = E2.dir')
}
```

(4) EJEMPLOS

```
ARRAY1: array[0..9] of integer;
ARRAY2: array[10..50] of float;

b, c, i: INTEGER
d: FLOAT

...
```

`b := 10;`	$\rightsquigarrow$	1001: t0 = 10 1002: b = t0
`c := 3 + ARRAY1[5]`	$\rightsquigarrow$	1003: t1 = 5 - 0 1004: t2 = t1 * 4 /*tamaño entero = 4*/ 1005: t3 = ARRAY1[t2] 1006: t4 = 3 + t3 1007: c = t4
`d := ARRAY2[i] + c * b`	$\rightsquigarrow$	1008: t5 = c * b 1009: t6 = i - 10 1010: t7 = t6 * 8 /*tamaño float = 8*/ 1011: t8 = ARRAY2[t7] 1012: t9 = t5 + t8 1013: d = t8

8.3.2 CI para expresiones lógicas

Dos posibilidades

1. Codificando valores *true/false* numéricamente
   - *true*$\neq$0, *false*=0
   - Evaluar expr. lógicas del mismo modo que las aritméticas
2. Mediante control de flujo (saltos)
   - El valor de una expresión booleana se representa implícitamente mediante una posición alcanzada en el programa
     $\longrightarrow$ si *true* salta a un punto; si *false* salta a otro
   - Ventajoso para evaluar exptr. boolenas en instrucciones de control de flujo
     $\longrightarrow$ facilita evaluar expresiones lógicas en cortocircuito

(Veremos sólo la primera posibilidad)

$E \longrightarrow E$ **op** E
(OR, AND, XOR)
	`{ E0.dir = crearTemporal()`
	`  E0.codigo = E1.codigo +`
	`  E2.codigo +`
	`  generarCI('E0.dir = E1.dir OP E2.dir')`
	`}`

$E \longrightarrow$ **op** E
(NOT, SHIFT, ...)
	`{ E0.dir = crearTemporal()`
	`  E0.codigo = E1.codigo +`
	`  generarCI('E0.dir = OP E1.dir')`
	`}`

8.3.2 (cont...)

$E \longrightarrow E$**op_rel**E $(=, <, >, !=, ...)$	{ E0.dir = crearTemporal() ETQ_TRUE = nuevaEtiqueta() ETQ_FIN = nuevaEtiqueta() E0.codigo = E1.codigo + E2.codigo + generarCI('if (E1.dir OPREL E2.dir) goto ETQ_TRUE') generarCI('E0.dir = 0')+ generarCI('goto ETQ_FIN)+ generarCI('ETQ_TRUE:')+ generarCI('E0.dir = 1')+ generarCI('ETQ_FIN:') }
$E \longrightarrow$ **true**	{ E.dir = crearTemporal() E.codigo = generarCI('E.dir = 1') }
$E \longrightarrow$ **false**	{ E.dir = crearTemporal() E.codigo = generarCI('E.dir = 0') }

8.3.3 Instrucciones de control de flujo

Versión sencilla, considerando expr. booleanas con valores numéricos

Uso de etiquetas para dar soporte al control de flujo en el C.I.

SALTOS

if *expresion* **then** *sentencias*

if *expresion* **then** *sentencias1* **else** *sentencias2*

BUCLES

while *expresion* **do** *sentencias*

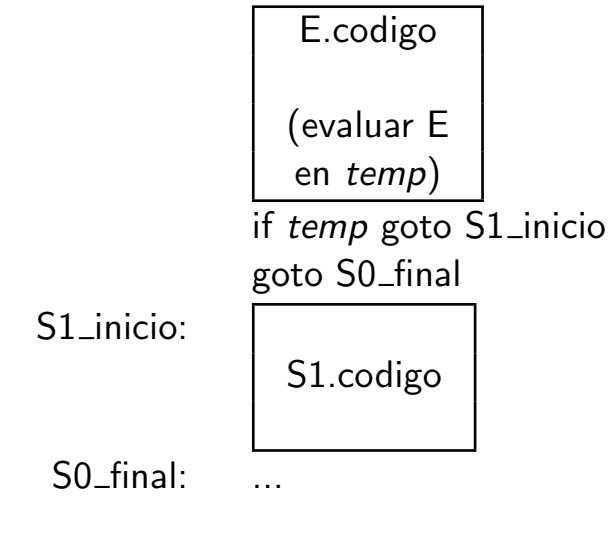
for *identificador* := *expresion1* **to** *expresion2* **do** *sentencias*

EXTENSIONES

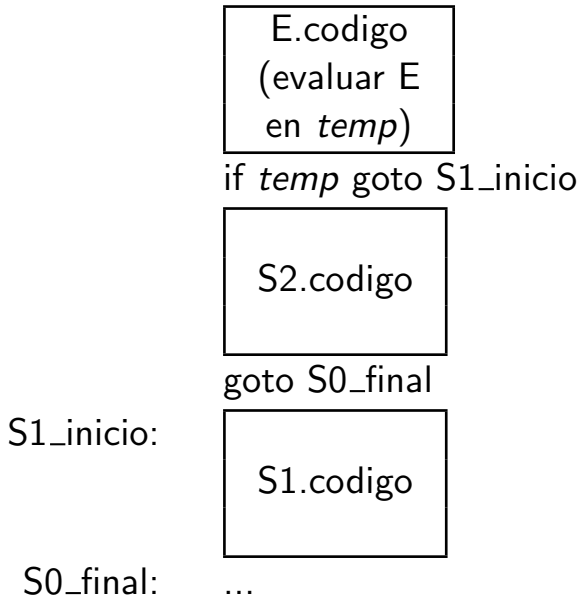
switch *expresion*
begin
case *valor1* : *sentencias1*
case *valor2* : *sentencias2*
...
case *valorN* : *sentenciasN*
default *sentenciasD*
end

8.3.3 (cont...) SALTOS

if - then



if - then - else



$S \rightarrow$ **if** E **then** S

```
{ S1_inicio = nuevaEtiqueta()
  S0_final  = nuevaEtiqueta()
  S0.codigo = E.codigo+
        generarCI('if E.dir goto S1.inicio')+
        generarCI('goto S0_final)+
        generarCI('S1_inicio:')+
        S1.codigo+
        generarCI('S0_final:')
}
```

$S \rightarrow$ **if** E **then** S **else** S

```
{ S1_inicio = nuevaEtiqueta()
  S0_final  = nuevaEtiqueta()
  S0.codigo = E.codigo +
        generarCI('if E.dir goto S1_inicio')+
        S2.codigo+
        generarCI('goto S0_final)+
        generarCI('S1_inicio:')+
        S1.codigo+
        generarCI('S0_final:')
}
```

8.3.3 (cont...) BUCLES

while

S0_inicio: E.codigo (evaluar E en *temp*)
if *temp* goto S1_inicio
goto S0_final
S1_inicio: S1.codigo
goto S0_inicio
S0_final: ...

for ascendente (usa nueva etiqueta TEST)

E1.codigo (límite inferior)
E2.codigo (límite superior)
id = E1
S0_test: if (id <= E2) goto S1_inicio
goto S0_final
S1_inicio: S1.codigo
id = id + 1
goto S0_test
S0_final: ...

$S \rightarrow$ **while** E **do** S

```
{ S0_inicio = nuevaEtiqueta()
  S0_final  = nuevaEtiqueta()
  S1_inicio = nuevaEtiqueta()
  S0.codigo = generarCI('S0_inicio:')+
     E.codigo+
     generarCI('if E.dir goto S1_inicio')+
     generarCI('goto S0_final)+
     generarCI('S1_inicio:')+
     S1.codigo+
     generarCI('goto S0_inicio)+
     generarCI('S0_final:')
}
```

$S \rightarrow$ **for** $id = E$ **to** E **do** S	{ S0_inicio = nuevaEtiqueta() S0_final = nuevaEtiqueta() S0_test = nuevaEtiqueta() S1_inicio = nuevaEtiqueta() S0.codigo = generarCI('S0_inicio:')+ E1.codigo + E2.codigo + generarCI('id.dir = E1.dir') generarCI('S0_test:')+ generarCI('if (id.dir <= E2.dir) goto S1_inicio')+ generarCI('goto S0_final)+ generarCI('S1_inicio:')+ S1.codigo+ generarCI('id.dir = id.dir + 1')+ generarCI('goto S0_test)+ generarCI('S0_final:') }

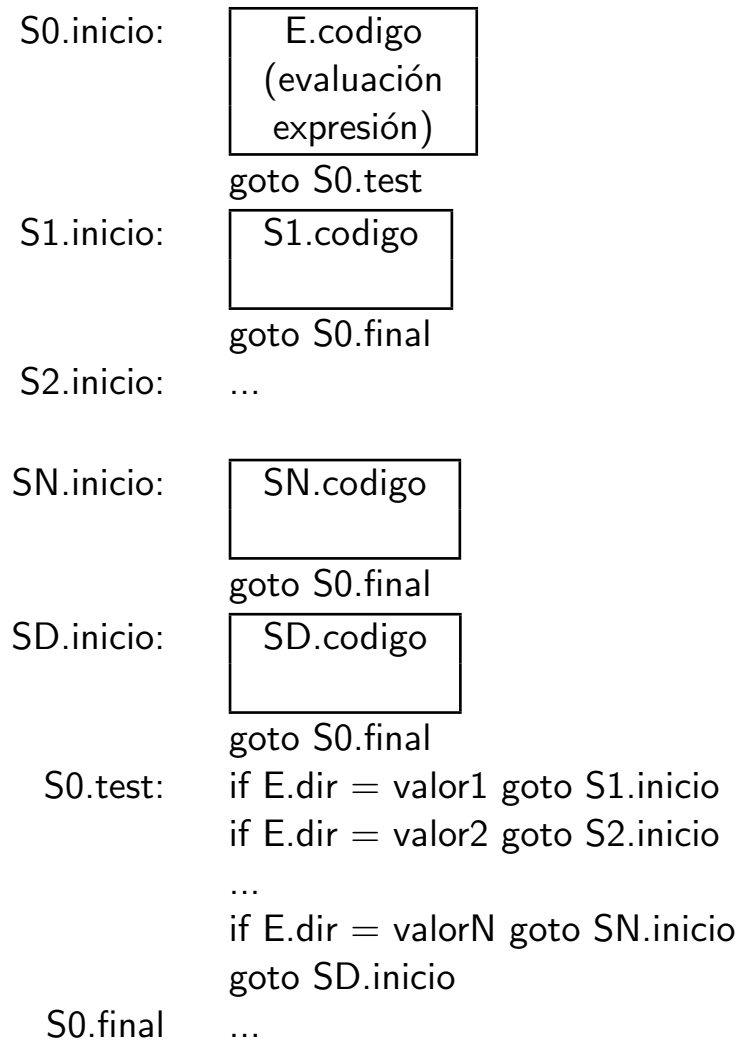
8.3.3 (cont...) CASOS ESPECIALES

Alteraciones de flujo dentro de bucles.

- Salida: (`exit, break`) **goto** a la etiqueta *S0.final*
- Cancelar iteración: (`continue`)
  → en **while**: **goto** al incio de la expresión de comparación (*S0.inicio*)
  → en **for**: **goto** a la instrucción de incremento

Instrucción CASE

switch *expresion*
  begin
    case *valor1* : *sentencias1*
    case *valor2* : *sentencias2*
    ...
    case *valorN* : *sentenciasN*
    default *sentenciasD*
  end



NOTAS

- En C, si no se incluye `break`, continúa la ejecución en el siguente caso.
  → se incluirá el `goto S0.final` en los casos donde exista **break**
  → consistente con comportamiento de **break** en **while** y **for** indicado anteriormente
- Si hubiera muchos casos, usar tabla de pares (caso, etiqueta)
  → la selección de caso se haría dentro de un bucle que iteraría sobre esa tabla

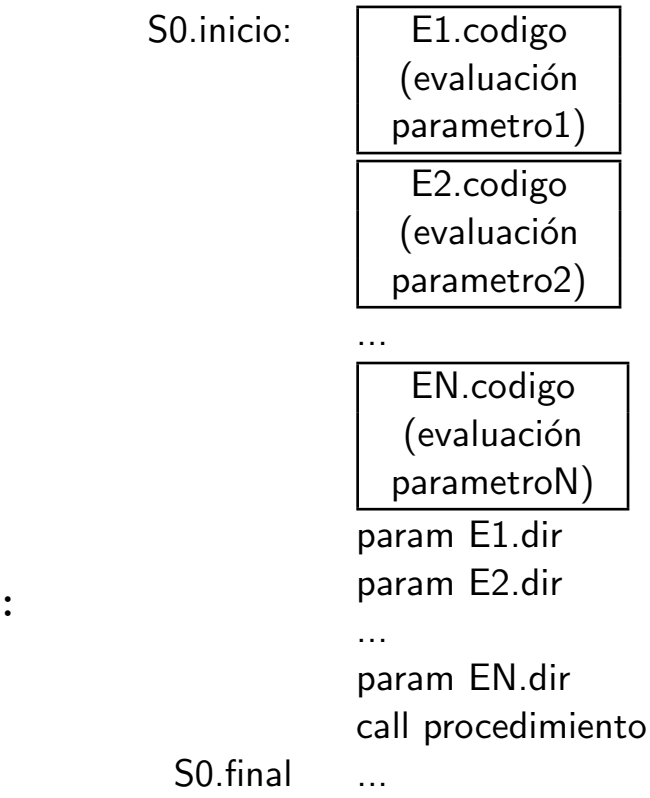
8.3.4 Procedimientos y Funciones

Esquema de llamada

```
procedimeinto(E, E, ... , E);
```

Ejemplos

```
imprimir(nombre, DNI, edad);
mostrar_resultados;
b := media(A[5], velocidad*15):
```



El uso de C.I. oculta las operacioens adicionales necesarias en la llamada y el retorno de procedimientos.

- LLAMADA
  - Evaluar valores de los parámetros
  - Crear el *registro de actvación* asociado al nuevo procedimiento
  - Paso de parámetros
  - Guardar estado del procedimeinto actual
  - Guardar dirección de retorno
  - Salto al inicio del código del procedmiento llamado
- RETORNO
  - Recuperar estado del procedimiento llamador
  - Salto a siguiente instrucción
  - Recuperar valor devuelto

Secuencias de llamada/retorno: alta dependencia de la máquina objetivo

- convenciones de llamada + instrucciones específicas