Capítulo 6

Generación de código

Así como el hierro se oxida por falta de uso,
también la inactividad destruye el intelecto

Leonardo Da Vinci

Resumen: En este tema se presenta la máquina-P, una máquina de pila típica para compilar a ella lenguajes imperativos, y se dan los esquemas de traducción para las construcciones más relevantes de los mismos, incluyendo el paso de parámetros y las llamadas a procedimiento. Se espera que el alumno sepa generar código manualmente a partir de estos esquemas.

6.1. Traducción de lenguajes imperativos. La máquina-P

⋆ El modelo de cómputo de un lenguaje imperativo consiste esencialmente en la existencia de un **estado**, que es transformado sucesivamente mediante la ejecución de **instrucciones**. El programa comienza en un estado **inicial**, y cada intrucción modifica el estado en curso y produce un nuevo estado. Las instrucciones se ejecutan **secuencialmente** en el orden especificado por el programador.

⋆ Hay dos tipos de instrucccciones:

- Las que modifican propiamente el estado. La más relevante es la **instrucción de asignación**.
- Las que evalúan el estado y en función del mismo dirigen el **flujo de control** del programa.

⋆ El estado está formado por el valor de las **variables** del programa en cada instante de la ejecución. Las variables pueden ser de tipo simple o estructurado, tales como

vectores, matrices y registros. Al nivel del código máquina, tales variables son simplemente **direcciones de memoria**. Una parte de la generación de código consiste en **asignar** tales direcciones.

⋆ Las variables de los procedimientos se suelen asignar en la **pila**, tal como se vió en el Capítulo 4. La pila contiene en un instante dado un **marco de activación** por cada procedimiento activo de la cadena de llamadas en curso. El marco contiene las variables locales, los argumentos del procedimiento y algunos enlaces de gestión de la propia pila. Este esquema da cabida a las diferentes **encarnaciones** de las variables de un procedimiento recursivo.

⋆ Otras variables pueden ser asignadas en el **montón**, e incluso otras, con un tiempo de vida que abarque toda la ejecución del programa, pueden ser asignadas **estáticamente**.

⋆ Las expresiones del lenguaje, que aparecen en la parte derecha de las asignaciones y en las condiciones que evalúan las instrucciones de control de flujo, son términos **arborescentes**. Una forma cómoda de evaluarlas es realizando un recorrido en **postorden** del árbol con la ayuda de una **pila**:

- Cuando se evalúa una **hoja**, que corresponde a una constante o una variable, simplemente se apila su valor.
- Cuando se evalúa un **nodo interno**, que corresponde a un operador normalmente binario (lo denotaremos $\oplus$), se realiza la siguiente secuencia:
  1. Se desapila la cima y la subcima.
  2. Se operan sus valores de la siguiente forma: $subcima \oplus cima$.
  3. Se apila el valor resultante.

⋆ Es importante hacer notar que el tamáño máximo de esta pila para evaluar expresiones se puede determinar **estáticamente** a partir de la profundidad del árbol que representa la expresión.

⋆ La máquina objetivo de la compilación ha de suministrar también soporte para la traducción de las **instrucciones de control de flujo**. Las instrucciones máquina necesarias son las siguientes:

- Instrucciones de **salto incondicional**.
- Instrucciones de **salto condicional**.
- Instrucción de **salto a procedimiento** que salve en algún registro la **dirección de retorno**. Estas direcciones han de apilarse para poder volver ordenadamente de los procedimientos llamados.

⋆ Las máquinas reales presentan algunas complicaciones para poder realizar una compilación eficiente:

- Ofrecen para las variables tenerlas en **memoria central** o en **registros**. Las operaciones aritmético-lógicas suelen estar soportadas exclusivamente sobre registros. Los movimientos entre registros son dos o tres órdenes de magnitud más rápidos que los movimientos entre estos y la memoria.
- El número de registros, aunque suele ser elevado, es **finito**. Es necesario prever que, en una cierta compilación, no todas las variables locales podrán estar permanentemente en registros.
- Las máquinas son sensibles al **orden** en que se ejecutan las instrucciones de un **bloque básico** (un bloque basico es una secuencia que no incluye saltos). Ciertos órdenes permiten aprovechar mejor las distintas unidades funcionales de la unidad de proceso y aumentar el grado de paralelismo del programa, y por tanto su eficiencia.

⋆ Una alternativa sencilla es definir una **máquina virtual** que facilite la traducción de las construcciones imperativas y generar código para dicha máquina. La máquina virtual pueder ser **interpretada** con poco esfuerzo en cualquier máquina real. Otra posibilidad que llevaria algo más de esfuerzo, pero sería más eficiente, sería compilar el código virtual al código nativo de una máquina real.

⋆ Las máquinas virtuales ofrecen facilidad de compilación y **portabilidad** del lenguaje fuente, ya que la mayor parte del trabajo realizado por el compilador es **independiente** de una máquina real concreta.

⋆ En este tema hemos elegido este enfoque y como máquina virtual la **máquina-P**, diseñada por Niklaus Wirth hacia 1970 en la Universidad de Zurich para soportar la compilación y la portabilidad de su lenguaje **Pascal**. En ella se han inspirado todas las máquinas virtuales posteriores, como la que dio soporte a Smalltalk (1981), o la que da soporte a Java (1992).

⋆ La máquina-P distingue dos zonas dedicadas al programa en ejecución:

- `CODE`, vector de instrucciones-P que componen el programa. Un registro especial `PC` (*program counter*) indica la siguiente instrucción `CODE[PC]` a ejecutar por la máquina.
- `STORE`, **pila** de datos. En ella se encuentran los marcos de activación de la cadena de llamadas en curso. La parte final de la pila también se usa para la evaluación de expresiones. La pila crece hacia las direcciones altas. En la posición 0 se halla el marco de activación del programa principal. Este marco nunca es desapilado. Un registro especial `SP` (*stack pointer*) indica la última posición ocupada, es decir la cima de la pila `STORE[SP]`.
- La parte alta del vector `STORE` está dedicada al **montón**, o zona de memoria reservada y liberada dinámicamente. La pila y el montón crecen la una al encuentro del otro mientras reservan memoria libre. Un registro `NP` (*new pointer*) indica la última posición ocupada por el montón `STORE[NP]`. Si se produce una

colisión (condición `SP` $\geq$ `NP`), el programa no puede seguir ejecutándose por falta de memoria.

⋆ El ciclo básico del intérprete de la máquina-P tiene el siguiente aspecto:

```
loop
  PC := PC + 1;
  interpretar instrucción CODE[PC-1];
end loop
```

Nótese que la instrucción ejecutada puede modificar el valor de `PC` y por tanto no sería correcto incrementar `PC` despues de ejecutarla.

⋆ La máquina-P soporta dos tipos numéricos, entero (i) y real (r), el tipo booleano (b), y el tipo *dirección* de memoria (a). Algunas instrucciones tienen varias versiones según el tipo de los operandos. Así $\texttt{add}_i$ denota la suma de dos enteros y $\texttt{add}_r$ la de dos reales. Los valores de estos tipos, así como una instrucción-P, ocupan **una palabra** de la maquina-P, cuya longitud no se especifica.

6.2. Traducción de expresiones y asignaciones

⋆ En la Figura 6.1 se muestran las instrucciones de la máquina-P que se ocupan de las operaciones aritméticas, lógicas y relacionales y se da su significado. Las que tienen dos operandos responden al esquema de desapilar la cima y subcima de la pila, realizar la operación $\mathit{subcima} \oplus \mathit{cima}$ y apilar el resultado. Las de un operando pueden entenderse como si operaran sobre la cima, o como que desapilan la cima, operan con ella y apilan el resultado.

⋆ En la Figura 6.2 se muestran las instrucciones-P que apilan variables y constantes, o almacenan la cima de la pila en una variable. Lógicamente dichas variables están ubicadas en la propia pila. Por el momento utilizamos desplazamientos **absolutos** q tomados desde el comienzo de la pila. Más adelante, las direcciones serán relativas al marco de activación en que se encuentra la variable y tendrán en cuenta el nivel de **anidamiento** estático de la variable accedida con respecto al bloque que contiene la aparición de uso.

⋆ Los esquemas de generación de código utilizarán de momento tres funciones mutuamente recursivas:

$\mathit{code}_R(\mathit{expresion})$	Genera código para expresiones
$\mathit{code}_L(\mathit{variable})$	Genera código para la parte izquierda de una asignación
$\mathit{code}(\mathit{instruccion})$	Genera código para instrucciones

⋆ De momento consideramos asignaciones de la forma `x := e`, donde `x` es una variable simple de la que se conoce su dirección absoluta y `e` es una expresión donde intervienen variables simples de las mismas características.

Instr.	Meaning	Cond.	Result
add *N*	$STORE[SP-1] := STORE[SP-1] +_N STORE[SP]$; $SP := SP-1$	(*N*, *N*)	(*N*)
sub *N*	$STORE[SP-1] := STORE[SP-1] -_N STORE[SP]$; $SP := SP-1$	(*N*, *N*)	(*N*)
mul *N*	$STORE[SP-1] := STORE[SP-1] *_N STORE[SP]$; $SP := SP-1$	(*N*, *N*)	(*N*)
div *N*	$STORE[SP-1] := STORE[SP-1] /_N STORE[SP]$; $SP := SP-1$	(*N*, *N*)	(*N*)
neg *N*	$STORE[SP] := -_N STORE[SP]$	(*N*)	(*N*)
and	$STORE[SP-1] := STORE[SP-1]$ *and* $STORE[SP]$; $SP := SP-1$	(b, b)	(b)
or	$STORE[SP-1] := STORE[SP-1]$ *or* $STORE[SP]$; $SP := SP-1$	(b, b)	(b)
not	$STORE[SP] :=$ *not* $STORE[SP]$	(b)	(b)
equ *T*	$STORE[SP-1] := STORE[SP-1] =_T STORE[SP]$; $SP := SP-1$	(*T*, *T*)	(b)
geq *T*	$STORE[SP-1] := STORE[SP-1] \geq_T STORE[SP]$; $SP := SP-1$	(*T*, *T*)	(b)
leq *T*	$STORE[SP-1] := STORE[SP-1] \leq_T STORE[SP]$; $SP := SP-1$	(*T*, *T*)	(b)
les *T*	$STORE[SP-1] := STORE[SP-1] <_T STORE[SP]$; $SP := SP-1$	(*T*, *T*)	(b)
grt *T*	$STORE[SP-1] := STORE[SP-1] >_T STORE[SP]$; $SP := SP-1$	(*T*, *T*)	(b)
neq *T*	$STORE[SP-1] := STORE[SP-1] \neq_T STORE[SP]$; $SP := SP-1$	(*T*, *T*)	(b)

Figura 6.1: Instrucciones aritméticas, lógicas y relacionales de la máquina-P

Instr.	Meaning	Cond.	Result
ldo *T q*	*SP* := *SP*+1; *STORE*[*SP*] := *STORE*[*q*]	*q* ∈ [0,*maxstr*]	(*T*)
ldc *T q*	*SP* := *SP*+1; *STORE*[*SP*] := *q*	Type(*q*) = *T*	(*T*)
ind *T*	*STORE*[*SP*] := *STORE*[*STORE*[*SP*]]	(a)	(*T*)
sro *T q*	*STORE*[*q*] := *STORE*[*SP*]; *SP* := *SP*−1	(*T*) *q* ∈ [0,*maxstr*]	
sto *T*	*STORE*[*STORE*[*SP*−1]] := *STORE*[*SP*]; *SP* := *SP*−2	(a, *T*)	

Figura 6.2: Instrucciones de carga y almacenamiento de la máquina-P

⋆ En la Figura 6.3 se muestran los esquemas de generación para expresiones, parte izquierda de una asignación y asignaciones completas. Un invariante importante para razonar sobre la corrección de $code_R(e)$ es que esta función genera un código para e que, cuando se ejecute, incrementará la pila en una palabra con respecto al estado de partida, y dicha palabra contendrá el valor de la expresión. La demostración es una simple inducción sobre la estructura de las expresiones:

- Es fácil comprobar que los casos base (constante, variable) cumplen la propiedad.
- Haciendo la hipótesis de inducción de que las llamadas recursivas cumplen la propiedad, la llamada externa $code_R(e_1 \oplus e_2)$ también la cumple.

⋆ Nótese que los esquemas utilizan un parámetro adicional ρ que proporciona las direcciones absolutas de las variables. Este **entorno de direcciones** se supone creado previamente durante la fase de asignación de memoria.

⋆ En la Figura 6.4 se muestra un ejemplo completo de generación de código para la asignación $a := (b + (b * c))$ suponiendo $\rho = [a \mapsto 5, b \mapsto 6, c \mapsto 7]$.

6.3. Traducción de instrucciones de control

⋆ En la Figura 6.5 se muestran las dos instrucciones de salto que van a dar soporte a la traducción de las instrucciones de control de Pascal. Nótese que las direcciones de código son absolutas con respecto al comienzo del vector `CODE`. Nótese también que la instrucción de salto condicional "consume" el valor booleano en la cima de la pila.

Function		Condition
$code_R(e_1 = e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **equ** T	Type(e_1) = Type(e_2) = T
$code_R(e_1 \neq e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **neq** T	Type(e_1) = Type(e_2) = T
⋮		
$code_R(e_1 + e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **add** N	Type(e_1) = Type(e_2) = N
$code_R(e_1 - e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **sub** N	Type(e_1) = Type(e_2) = N
$code_R(e_1 * e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **mul** N	Type(e_1) = Type(e_2) = N
$code_R(e_1/e_2)\ \rho$	$= code_R\ e_1\ \rho;\ code_R\ e_2\ \rho;$ **div** N	Type(e_1) = Type(e_2) = N
$code_R(-e)\ \rho$	$= code_R\ e\ \rho;$ **neg** N	Type(e) = N
$code_R\ x\ \rho$	$= code_L\ x\ \rho;$ **ind** T	x variable identifier of type T
$code_R\ c\ \rho$	= **ldc** $T\ c$	c constant of type T
$code(x := e)\ \rho$	$= code_L\ x\ \rho;\ code_R\ e\ \rho;$ **sto** T	x variable identifier
$code_L\ x\ \rho$	= **ldc** a $\rho(x)$	x variable identifier

Figura 6.3: Esquemas de generación de código para expresiones y asignaciones

$code(a := (b + (b * c)))\ \rho$
= $code_L\ a\ \rho;$ $code_R(b + (b * c))\ \rho;$ **sto** i
= **ldc** a 5; $code_R(b + (b * c))\ \rho;$ **sto** i
= **ldc** a 5; $code_R(b)\ \rho;$ $code_R(b * c)\ \rho;$ **add** i; **sto** i
= **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b * c)\ \rho;$ **add** i; **sto** i
= **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b)\ \rho;$ $code_R(c)\ \rho;$ **mul** i; **add** i; **sto** i
= **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_R(c)\ \rho;$ **mul** i; **add** i; **sto** i
= **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; **ldc** a 7; **ind** i; **mul** i; **add** i; **sto** i

Figura 6.4: Ejemplo de generación de código para una asignación

Instr.	Meaning	Comments	Cond.	Result
ujp q	$PC := q$	Unconditional branch	$q \in$ [0,*codemax*]	
fjp q	**if** $STORE[SP] = false$	Conditional branch	(b)	
	then $PC := q$		$q \in$ [0,*codemax*]	
	fi			
	$SP := SP - 1$			

Figura 6.5: Instrucciones-P de salto condicional e incondicional

⋆ En los esquemas que siguen, admitimos la posibilidad de generar **etiquetas** que podrán ser utilizadas como destino de las instrucciones de salto. Se trata de una simplificación del esquema, el cual realmente calcularía el desplazamiento en número de instrucciones de la etiqueta con respecto al comienzo del bloque de código, y después le sumaría la dirección absoluta del comienzo de dicho bloque para calcular la dirección absoluta q a la cual hay que saltar y que ha de incluir como argumento en la instrucción de salto.

⋆ Un invariante del esquema *code* es que la ejecución de las instrucciones que genera no modifica el tamaño de la pila.

⋆ Presentamos el esquema de traducción de la instrucción **if** con dos ramas:

$$
\begin{array}{rl}
code(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2)\ \rho = & code_R\ e\ \rho; \\
& \textbf{fjp } l_1; \\
& code\ s_1\ \rho; \\
& \textbf{ujp } l_2; \\
l_1: & code\ s_2\ \rho; \\
l_2: &
\end{array}
$$

⋆ El de la instrucción **if** con una rama:

$$
\begin{array}{rl}
code(\textbf{if } e \textbf{ then } s)\ \rho = & code_R\ e\ \rho; \\
& \textbf{fjp } l; \\
& code\ s\ \rho; \\
l: &
\end{array}
$$

⋆ El de la instrucción **while**

$$\begin{array}{rll} code(\textbf{while}\ e\ \textbf{do}\ s)\ \rho = & l_1: & code_R\ e\ \rho; \\ & & \textbf{fjp}\ l_2; \\ & & code\ s\ \rho; \\ & & \textbf{ujp}\ l_1; \\ & l_2: & \end{array}$$

⋆ El de la instrucción **repeat**:

$$\begin{array}{rll} code(\textbf{repeat}\ s\ \textbf{until}\ e)\ \rho = & l: & code\ s\ \rho; \\ & & code_R\ e\ \rho; \\ & & \textbf{fjp}\ l; \end{array}$$

⋆ Y el de la composición secuencial de dos instrucciones:

$$code\ (s_1; s_2)\ \rho = code\ s_1\ \rho; code\ s_2\ \rho$$

⋆ En la Figura 6.6 (a) se muestra el código generado para la instrucción:

$$\textbf{if}\ a > b\ \textbf{then}\ c := a\ \textbf{else}\ c := b$$

Y en la Figura 6.6 (b) el código para:

$$\textbf{while}\ a > b\ \textbf{do}\ \{c := c + 1; a := a - b\}$$

Suponemos de nuevo el entorno $\rho = [a \mapsto 5, b \mapsto 6, c \mapsto 7]$.

⋆ La traducción de la instrucción **case** de Pascal es algo más compleja. El objetivo de esta instrucción es bifurcar en **tiempo constante** a una entre varias instrucciones etiquetadas, según el valor entero de una expresión de control calculada en tiempo de ejecución. Las etiquetas son valores numéricos constantes y disjuntos. La sintaxis de **case** permite especificar rangos de valores, dejar huecos en la numeración y tener etiquetas numéricamente no consecutivas. En aras de la claridad, presentamos sólo el esquema básico que traduce instrucciones **case** de la forma:

```
case e of
 0: S0;
 ...
 k: Sk;
end
```

donde las etiquetas van consecutivamente de 0 a k sin dejar huecos

⋆ Para dar soporte a la traducción, se necesita una instrucción que calcule un salto diferente según el valor entero que encuentre en la cima de la pila. Es la siguiente:

$$\textbf{ixj}\ q \equiv \{\texttt{PC} := \texttt{STORE}[\texttt{SP}] + q; \texttt{SP} := \texttt{SP} - 1\}$$

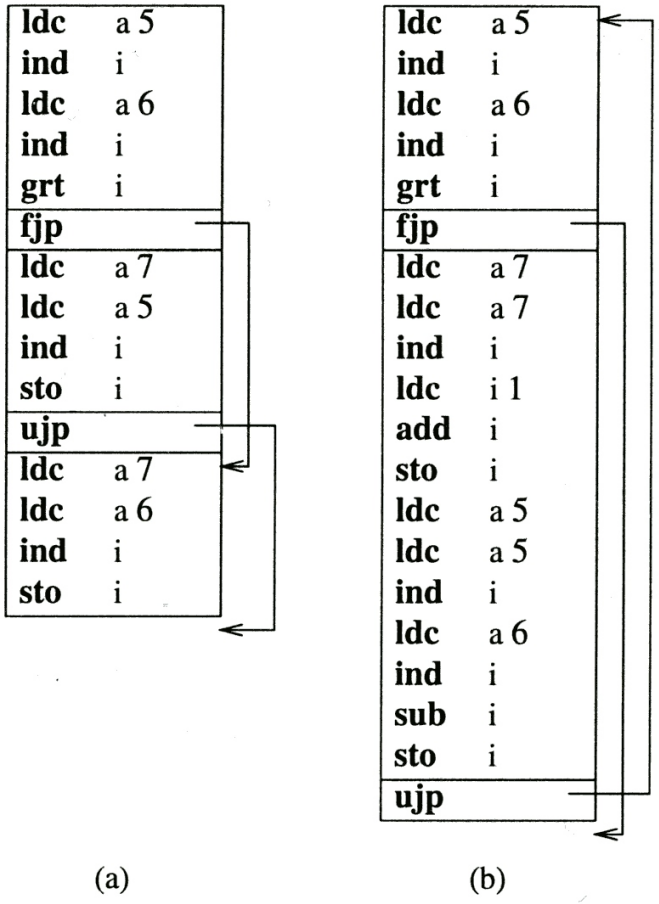

Figura 6.6: Ejemplos de generación de código con instrucciones de control

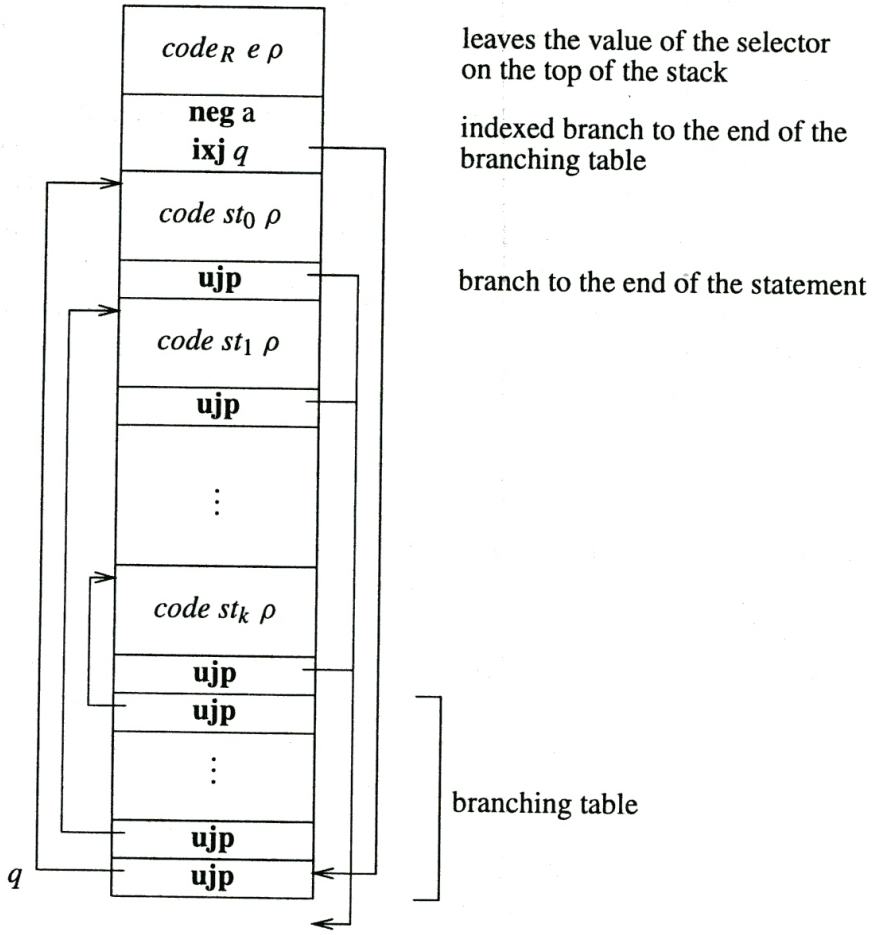

Figura 6.7: Generación de código para la instrucción **case** básica

⋆ El esquema de traducción del **case** anterior se muestra en la Figura 6.7 en forma gráfica. La parte final de la traducción recibe el nombre de **tabla de saltos**. Nótese que el acceso a cualquier instrucción etiquetada requiere siempre dos saltos, es decir un tiempo que no depende del número de alternativas.

6.4. Asignación de memoria

⋆ Los compiladores optimizan el uso de la memoria y suelen empaquetar varios valores "pequeños" en una palabra de la máquina. Asímismo, los valores en coma flotante de doble precisión pueden ocupar más de una palabra hardware. Algunos lenguajes especifican tipos enteros de distinta longitud (típicamente entre 1 y 8 octetos). Por otra parte, las instrucciones máquina de carga y almacenamiento pueden cargar desde

un octeto a 8 siempre que la secuencia de octetos esté propiamente **alineada** en memoria. Por ejemplo, una instrucción que cargue 4 octetos desde memoria, exige que el primer octeto empiece en una dirección múltiplo de cuatro (acabada en dos ceros binarios). Estas restricciones complican la asignación de direcciones de memoria a las variables del programa.

⋆ Aquí haremos la suposición simplificadora de que cada valor básico (booleano, entero, real, puntero, dirección) ocupa **una palabra** y que no hay restricciones de alineamiento. Por otra parte, las direcciones serán **relativas** al comienzo del bloque de activación del procedimiento que contiene las declaraciones. Por razones que se aclararán más adelante, las palabras 0 a 4 de un bloque están reservadas.

⋆ Supongamos la siguiente declaración de variables de tipos simples en la cabecera de un procedimiento:

$$\textbf{var } v_0 : t_0;\ \ldots\ ; v_n : t_n$$

La **función de asignación de memoria** es simplemente: $\rho(v_i) = 5 + i,\ 0 \leq i \leq n$.

⋆ Si los tipos t_i no son simples, exigiremos que su tamaño pueda ser determinado **estáticamente**. Denotamos por $size(t)$ el número de palabras ocupado por una variable de tipo t. En ese caso, la asignación de memoria para la anterior declaración se convierte en:

$$\rho(v_i) = 5 + \sum_{j=0}^{i-1} size(t_j),\quad 0 \leq i \leq n$$

6.4.1. Vectores y matrices estáticos

⋆ Una matriz estática de k dimensiones en Pascal se declara:

$$\textbf{var } b : \textbf{array } [u_1..o_1, \ldots, u_k..o_k] \textbf{ of } t$$

donde $u_i,\ o_i,\ 1 \leq i \leq k$, son constantes conocidas en tiempo de compilación. Su tamaño se calcula multiplicando el tamaño de sus dimensiones:

$$size(\textbf{array } [u_1..o_1, \ldots, u_k..o_k] \textbf{ of } t) = \left(\prod_{i=1}^{k} (o_i - u_i + 1)\right) \times size(t)$$

⋆ Hay dos modos de disponer matrices en memoria: por filas y por columnas. En la disposición por filas se hace variar más rápidamente el índice k. Cada vez que este completa un ciclo, se hace variar en uno el índice $k-1$ y así sucesivamente. El índice que varía más lentamente es el 1 (el de las "filas"). Podemos considerar que la memoria contiene una secuencia de $o_1 - u_1 + 1$ bloques, correspondientes a las filas u_1 a o_1,

cada uno consistente en una matriz donde se ha suprimido el índice 1, es decir de tipo **array** $[u_2..o_2, \ldots, u_k..o_k]$ **of** t, cuyo tamaño es:

$$\left(\prod_{i=2}^{k}(o_i - u_i + 1)\right) \times size(t)$$

Cada uno de estos bloques se puede descomponer del mismo modo como una "fila" de matrices con una dimensión menos, y así sucesivamente hasta llegar a los elementos individuales.

⋆ Suponiendo conocida la dirección $\rho(b)$ donde comienza la matriz, nos planteamos el problema de calcular la dirección del elemento $b[i_1, \ldots, i_k]$ donde $i_1, \ldots, i_k$ son expresiones enteras cuyo valor se conoce en tiempo de ejecución. Definimos las siguientes cantidades estáticas auxiliares:

$$\begin{array}{ll} d_j = o_j - u_j + 1 & 1 \leq j \leq k \\ d^j = \prod_{l=j+1}^{k} d_l & 1 \leq j \leq k \\ d = \sum_{j=1}^{k} u_j \times d^j & \\ g = size(t) & \end{array}$$

⋆ Llamaremos $\bar{i}_j$ al valor en tiempo de ejecución de la expresión entera i_j. Entonces, la dirección del elemento $b[i_1, \ldots, i_k]$ es:

$$\begin{array}{rcl} \rho(b) & + & (\bar{i}_1 - u_1) \times d_2 \times d_3 \times \ldots \times d_k \times size(t) + \\ & & (\bar{i}_2 - u_2) \times d_3 \times d_4 \times \ldots \times d_k \times size(t) + \\ & & \ldots \\ & & (\bar{i}_{k-1} - u_{k-1}) \times d_k \times size(t) + \\ & & (\bar{i}_k - u_k) \times size(t) \\ & = & \sum_{j=1}^{k}(\bar{i}_j - u_j) \times d^j \times g \\ & = & \sum_{j=1}^{k} \bar{i}_j \times d^j \times g - d \times g \end{array}$$

⋆ En definitiva, se trata de acumular una suma de productos de cantidades dinámicas (las $\bar{i}_j$) por cantidades estáticas, y finalmente restar una cantidad estática. La máquina-P suministra dos instrucciones de soporte:

$$\begin{array}{rcl} \textbf{ixa } q & \equiv & \texttt{STORE[SP} - 1\texttt{]} := \texttt{STORE[SP} - 1\texttt{]} + \texttt{STORE[SP]} \times q; \\ & & \texttt{SP} := \texttt{SP} - 1 \\ \textbf{dec } q & \equiv & \texttt{STORE[SP]} := \texttt{STORE[SP]} - q \end{array}$$

⋆ El esquema $code_L$ que genera código para calcular la dirección de un elemento de una matriz es el siguiente:

$$
\begin{array}{lcl}
code_L\ b[i_1,\ldots,i_k]\ \rho & = & \textbf{ldc}\ \rho(b); code_I\ [i_1,\ldots,i_k]\ g\ \rho \\
code_I\ [i_1,\ldots,i_k]\ g\ \rho & = & code_R\ i_1\ \rho; \textbf{ixa}\ (g \cdot d^1); \\
 & & code_R\ i_2\ \rho; \textbf{ixa}\ (g \cdot d^2); \\
 & & \ldots \\
 & & code_R\ i_k\ \rho; \textbf{ixa}\ (g \cdot d^k); \\
 & & \textbf{dec}\ (g \cdot d)
\end{array}
$$

⋆ El compilador de Pascal puede incluir opcionalmente comprobaciones en tiempo de ejecución de que los valores utilizados como índices están dentro de los límites inferior y superior de la dimensión correspondiente. La siguiente instrucción da soporte a esta necesidad:

$$
\begin{array}{lcl}
\textbf{chk}\ p\ q & \equiv & \text{si } (\texttt{STORE[SP]} < p) \vee (\texttt{STORE[SP]} > q) \\
 & & \text{entonces } \textit{error "value out of range"}
\end{array}
$$

⋆ El esquema $code_I$ con comprobaciones de rango es:

$$
\begin{array}{lcl}
code_I\ [i_1,\ldots,i_k]\ g\ \rho & = & code_R\ i_1\ \rho; \textbf{chk}\ u_1\ o_1; \textbf{ixa}\ (g \cdot d^1); \\
 & & code_R\ i_2\ \rho; \textbf{chk}\ u_2\ o_2; \textbf{ixa}\ (g \cdot d^2); \\
 & & \ldots \\
 & & code_R\ i_k\ \rho; \textbf{chk}\ u_k\ o_k; \textbf{ixa}\ (g \cdot d^k); \\
 & & \textbf{dec}\ (g \cdot d)
\end{array}
$$

6.4.2. Vectores y matrices dinámicos

⋆ En versiones posteriores de Pascal y en muchos otros lenguajes, es posible definir matrices cuyo tamaño es **dinámico**:

- el número k de dimensiones es estático.
- el tipo t de los elementos y su tamaño g son estáticos.
- los límites inferior y superior de cada dimensión, y por tanto el rango de la dimensión, pueden depender de parámetros recibidos en tiempo de ejecución.

⋆ Si el procedimiento tuviera declarada a lo sumo una matriz dinámica, podría asignarse al final del marco de activación y seguiría siendo posible una asignación estática de direcciones de memoria. Pero esta restricción no es razonable, con lo cual la existencia de matrices dinámicas aparentemente imposibilita poder hacer una asignación estática de direcciones.

⋆ Para resolver este problema, se recurre tener un **descriptor** de tamaño estático para cada matriz dinámica. En la Figura 6.8 se muestra una posible estructura de este descriptor, cuyo tamaño es de $3k+2$ palabras. La función ρ de asignación de memoria utiliza este tamaño como si fuera el tamaño de la matriz. De esta forma, la asignación de direcciones de memoria puede hacerse estáticamente.

Address	
0	Fictitious start address: a
1	Array size: i
2	Subtract for fictitious start address: i
3	u_1:i
4	o_1:i
⋮	⋮
$2k+1$	u_k:i
$2k+2$	o_k:i
$2k+3$	d_2:i
⋮	⋮
$3k+1$	d_k:i

Figura 6.8: Descriptor de una matriz dinámica de k dimensiones

⋆ El descriptor está inicialmente vacío. En tiempo de ejecución, al conocerse el rango de cada dimensión para todas las matrices dinámicas del procedimiento, es posible calcular su tamaño y todas las otras cantidades (las d_j, d^j, d, etc.). Antes de ejecutar el procedimiento se realizan las siguientes acciones:

- Se reserva memoria al final del marco de activación para cada matriz dinámica b y se calcula su dirección de comienzo $dir(b)$.
- Se rellenan las entradas del descriptor para b:
  - En la posición 0, se almacena $dir(b) - g \times d$.
  - En la posición 1, el tamaño $(\prod_{j=1}^{k} d_j) \times g$.
  - En la posición 2, la cantidad $g \times d$.
  - En las siguientes $2k$ posiciones, los límites u_j, o_j de cada dimensión.
  - En las últimas $k-1$ posiciones, las cantidades $d_2, \ldots, d_k$.

⋆ Las tres primeras posiciones del descriptor permiten realizar fácilmente copias de la matriz, en caso de que haya de ser pasada por valor o asignada a otra matriz igual.

⋆ Los límites u_j, o_j son útiles para realizar las comprobaciones de índices dentro de rango.

⋆ Una vez conocidos en tiempo de ejecución los valores $\bar{i}_j$ de los índices i_j, la generación de código utiliza la primera palabra del descriptor y las últimas $k-1$ palabras para componer el siguiente cálculo:

$$dir(b) - g \times d + ((\ldots((\bar{i}_1 \times d_2 + \bar{i}_2) \times d_3 + \bar{i}_3) \times d_4 + \ldots \bar{i}_{k-1}) \times d_k + \bar{i}_k) \times g$$

que da correctamente la dirección del elemento $b[i_1, \ldots, i_k]$.

⋆ La máquina-P suministra instrucciones especializadas para facilitar el cálculo de esta fórmula y el acceso al descriptor. No damos el detalle de las mismas ni el esquema por ser un tanto elaborados, pero la esencia es el cálculo descrito.

6.4.3. Registros

⋆ El tamaño de un tipo registro es la suma de los tamaños de los tipos de sus componentes. Si declaramos

$$\begin{array}{lll} \textbf{type } t & = & \textbf{record} \\ & & \quad c_1 : t_1; \\ & & \quad \ldots \\ & & \quad c_k : t_k \\ & & \textbf{end} \end{array}$$

entonces $size(t) = \sum_{j=1}^{k} size(t_j)$.

⋆ Si hacemos la suposición de que los nombres de los campos de un registro son disjuntos de los de cualquier otro registro y de los nombres del resto de las variables (siempre se puede conseguir asignando un identificador único a cada registro y adjuntando el identificador al nombre del campo), podemos utilizar la función ρ para asignar direcciones a los campos de un registro. Dichas direcciones serán **relativas** al comienzo del registro:

$$\rho(c_i) = \sum_{j=1}^{i-1} size(t_j), \quad 1 \leq i \leq k$$

⋆ Si tenemos una variable de tipo registro $v : t$, el esquema $code_L$ para calcular la dirección de un campo de un registro se define:

$$code_L(v.c_i) = \textbf{ldc}\ \rho(v);\ \textbf{inc}\ \rho(c_i)$$

donde el significado de la nueva instrucción-P es:

$$\textbf{inc}\ q \equiv \texttt{STORE[SP]} := \texttt{STORE[SP]} + q$$

6.4.4. Variables asignadas en el montón

⋆ En Pascal la instrucción **new**(p), donde p es un puntero que apunta a objetos de tipo t, reserva un bloque de memoria en el montón de un tamaño $size(t)$ y almacena en p la dirección de comienzo de dicho bloque. El tiempo de vida de este objeto termina, bien con una instrucción *dispose*(p), que no es estándar en Pascal, bien con una recolección de basura, si se detecta que el objeto no está vivo.

⋆ Como hemos dicho, el montón en la máquina-P crece hacia las direcciones bajas, y el registro `NP` marca la última posición ocupada. Para no tener que hacer el test de colisión con la pila cada vez que se modifica el registro `SP` (lo que ocurre con mucha frecuencia), existe otro registro `EP` (*extreme pointer*) que al entrar en un procedimiento contiene el valor máximo que puede alcanzar `SP` durante la ejecución del mismo. Así, el test se realiza una vez a la entrada del procedimiento, y cada vez que se modifica `NP`.

⋆ La instrucción-P de soporte también se llama **new**:

$$\begin{array}{lll} \textbf{new} & \equiv & \text{si } \texttt{NP} - \texttt{STORE[SP]} \leq \texttt{EP} \\ & & \text{entonces } \textit{error "mememe overflow"} \\ & & \text{si no} \\ & & \texttt{NP} := \texttt{NP} - \texttt{STORE}[SP]; \\ & & \texttt{STORE[STORE[SP} - 1]] := \texttt{NP}; \\ & & \texttt{SP} := \texttt{SP} - 2 \end{array}$$

que espera en la cima el tamaño del bloque y en la subcima la dirección del puntero en donde hay que almacenar la dirección del bloque reservado.

⋆ El esquema de traducción para la instrucción **new** de Pascal es:

$$code(\mathbf{new}(p))\ \rho = \mathbf{ldc}\ \rho(p); \mathbf{ldc}\ size(t);\ \mathbf{new}$$

donde p es una variable de tipo $\uparrow t$. Nótese que *code* cumple el invariante de que el tamaño final e inicial de la pila coinciden.

6.4.5. Esquema completo de traducción de expresiones izquierdas

⋆ En la parte izquierda de una asignación pueden aparecer expresiones arbitrariamente complejas que involucren accesos a elementos de una matriz, acceso indirecto a través de punteros y acceso a campos de registros, tales como:

$$x \uparrow [i+1, j+b[i]].a \uparrow [k].b$$

⋆ Tales accesos sofisticados también pueden aparecer en expresiones derechas, pero en ese caso aplicaremos el esquema de la Sección 6.2. Si e es una expresión izquierda:

$$code_R(e)\ \rho = code_L(e)\ \rho;\ \mathbf{ind}$$

⋆ Una expresión izquierda responde a la siguiente sintaxis:

$$\begin{array}{lcl} e & \rightarrow & x\,r \\ r & \rightarrow & .x\,r \\ & | & \uparrow r \\ & | & [i_1, \ldots, i_k]\,r \\ & | & \epsilon \end{array}$$

donde x denota un identificador arbitrario, e i_j son expresiones enteras.

⋆ El esquema completo de traducción de expresiones izquierdas es:

$$\begin{array}{lcl} code_L\ (x\,r)\ \rho & = & \mathbf{ldc}\ \rho(x); \\ & & code_M(r)\ \rho \\ code_M(.x\,r)\ \rho & = & \mathbf{inc}\ \rho(x); \\ & & code_M(r)\ \rho \\ code_M\ (\uparrow r)\ \rho & = & \mathbf{ind}; \\ & & code_M(r)\ \rho \\ code_M([i_1, \ldots, i_k]\,r)\ \rho & = & code_I[i_1, \ldots, i_k]\ g\ \rho; \\ & & code_M(r)\ \rho \\ code_M(\epsilon)\ \rho & = & \epsilon \end{array}$$

⋆ *Ejemplo.* Supongamos las siguientes declaraciones:

$$\begin{array}{lcl} \mathbf{type}\ t & = & \mathbf{record} \\ & & \quad a : \mathbf{array}\ [-5..5, 1..9]\ \mathbf{of}\ \mathbf{integer}; \\ & & \quad b :\uparrow t \\ & & \mathbf{end}; \\ \mathbf{var}\ i, j & : & \mathbf{integer}; \\ p & : & \uparrow t; \end{array}$$

y la expresión izquierda $p \uparrow .b \uparrow .a[i+1, j]$. Suponemos $\rho = [i \mapsto 5, j \mapsto 6, p \mapsto 7]$.

En primer lugar hacemos los cálculos:

$$
\begin{array}{ll}
size(t) = 11 \times 9 + 1 = 100 & d^2 = 1 \\
\rho(a) = 0 & d = (-5) \times 9 + 1 \times 1 = -44 \\
\rho(b) = 99 & g = 1 \\
d^1 = 9 &
\end{array}
$$

El código-P generado por $code_L(p \uparrow .b \uparrow .a[i+1, j])\ \rho$ es:

ldc 7;	$\rho(p)$
ind ;	$code_M(\uparrow \ldots)\ \rho$
inc 99;	$code_M(.b \ldots)\ \rho$
ind ;	$code_M(\uparrow \ldots)\ \rho$
inc 0;	$code_M(.a \ldots)\ \rho$
ldc 5;	$code_I([i+1, j])\ g\ \rho$
ind ;	
ldc 1;	
add ;	
ixa 9;	
ldc 6;	
ind ;	
ixa 1;	
dec (-44)	

6.5. Prodecimientos y paso de parámetros

⋆ Una **declaración** de procedimiento o función en Pascal consta de:

- Una **cabecera** con el nombre y la especificación de los parámetros formales, en la que se indica el tipo de cada uno y si se pasan **por valor** o **por referencia**.
- Un conjunto de **declaraciones locales**, que introducen procedimientos anidados, tipos, variables y constantes.
- Un **cuerpo ejecutable** compuesto por una o más instrucciones.

⋆ Una **activación** de procedimiento es una **instrucción** que contiene el nombre del procedimiento invocado y los argumentos reales de la invocación. Estos pueden ser:

- **Expresiones** del tipo apropiado, si corresponden a parámetros formales pasados por valor.
- **Variables** del tipo apropiado, si corresponden a parámetros formales pasados por referencia.

⋆ Una activación de **función** es similar salvo por el hecho de que es una **expresión** en lugar de una instrucción, y su valor es el valor devuelto por la función.

⋆ El código generado para una activación de procedimiento o función, denominado **secuencia de llamada**, produce los siguientes efectos:

1. **Apilar** un nuevo marco de activación encima del marco de activación en curso, que corresponde al procedimiento llamante.
2. Establecer los **enlaces estáticos y dinámicos** apropiados entre el nuevo marco y los marcos existentes en la pila.
3. **Evaluar** las expresiones correspondientes a los parámetros pasados por valor y **copiar** el valor resultante en el nuevo marco de activación, el cual tiene una (o varias) posición(es) reservada(s) para cada parámetro, según sea su tipo.
4. **Copiar** en el nuevo marco la dirección de los parámetros pasados por referencia.
5. **Saltar** a la primera instrucción ejecutable del procedimiento llamado, salvando en el nuevo marco la dirección de retorno.

⋆ Las primeras instrucciones que se ejecutan en el procedimiento llamado se denominan **secuencia de entrada**, y producen los siguientes efectos:

1. **Crear** las matrices dinámicas declaradas como variables locales y rellenar los campos de los descriptores correspondientes, que están en el propio marco de activación.
2. **Establecer** la cima `SP`, es decir el final del marco de activación, y `EP`, en función de la complejidad de las expresiones que evaluará el procedimiento.
3. **Comprobar** si la pila colisiona con el montón.
4. **Saltar** a la primera instrucción del cuerpo ejecutable

⋆ En la Figura 6.9 se muestra el formato del **marco de activación** del procedimiento en curso de ejecución. Las direcciones crecen hacia la derecha. Sus componentes son:

1. La palabra 0 está dedicada a devolver el **valor** de la función. Si se trata del marco de un procedimiento, esta palabra no se usa.
2. Las palabras 1 y 2 contienen respectivamente el **enlace estático** `SL` y el **enlace dinámico** `DL` del marco. El primero apunta a la primera palabra del marco del **predecesor estático** de este procedimiento, y el segundo apunta a la primera palabra del marco de su **predecesor dinámico**, es decir de su llamante.
3. La palabra 3 contiene el registro `EP` del llamante.
4. La palabra 4 contiene la **dirección de retorno** `RA`, es decir la dirección de código a la que hay que volver al terminar la llamada.
5. A partir de la palabra 5, se asignan las direcciones de los **parámetros formales**.
6. A continuación se reserva espacio para las **variables locales**.
7. La última parte del marco la ocupan las **matrices dinámicas**, cuyo tamaño no se conoce hasta el momento de llamada.

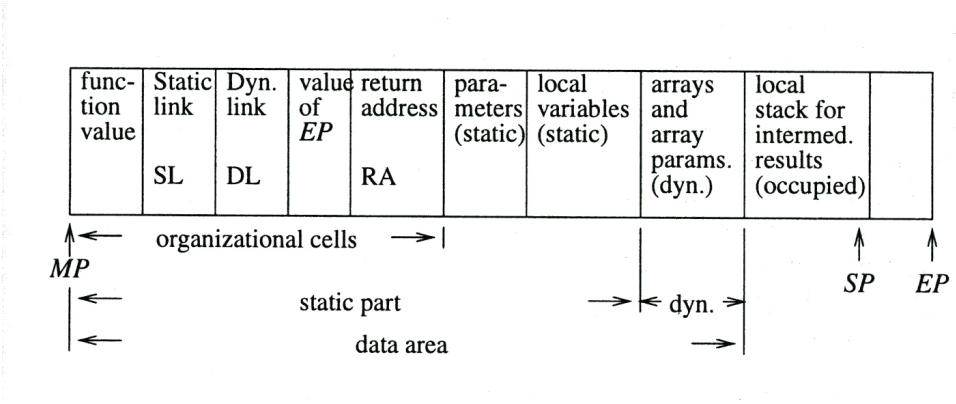

Figura 6.9: Marco de activación de un procedimiento o función

8. Aquí termina propiamente el marco de activación y comienza la **pila de evaluación de expresiones**, que puede crecer hasta el valor indicado por `EP`.
9. El registro `SP` marca como se ha visto la cima de la pila, y un nuevo registro `MP` (*mark pointer*) apunta al comienzo del marco de activación.

⋆ En la Figura 6.10 se muestra un programa, su ejecución, y la pila de marcos con los enlaces estáticos y dinámicos.

6.5.1. Direccionamiento de variables

⋆ Para explicar el **direccionamiento de variables** locales y no locales en tiempo de ejecución, introducimos el concepto de **profundidad de anidamiento** (abreviado pa), aplicado tanto a apariciones de definición como a apariciones de uso:

$$\begin{array}{ll} \text{apariciones de definición} & \begin{cases} pa(main) & = 0 \\ pa(id \text{ declarado en unidad con } pa = n) & = n+1 \end{cases} \\ \text{apariciones de uso} & \{\ pa(id \text{ usado en unidad con } pa = n) \quad = n+1 \end{array}$$

⋆ En la Figura 6.11 se muestran las apariciones de definición y se uso del programa de la Figura 6.10, indicando sus profundidades de anidamiento.

⋆ Para cada uso de una variable x debemos conocer:

- $pa(\text{ap. de uso de } x) = m$.
- $pa(\text{ap. de definición de } x) = n$.
- $\rho(x)$ en la unidad en que está declarada x.

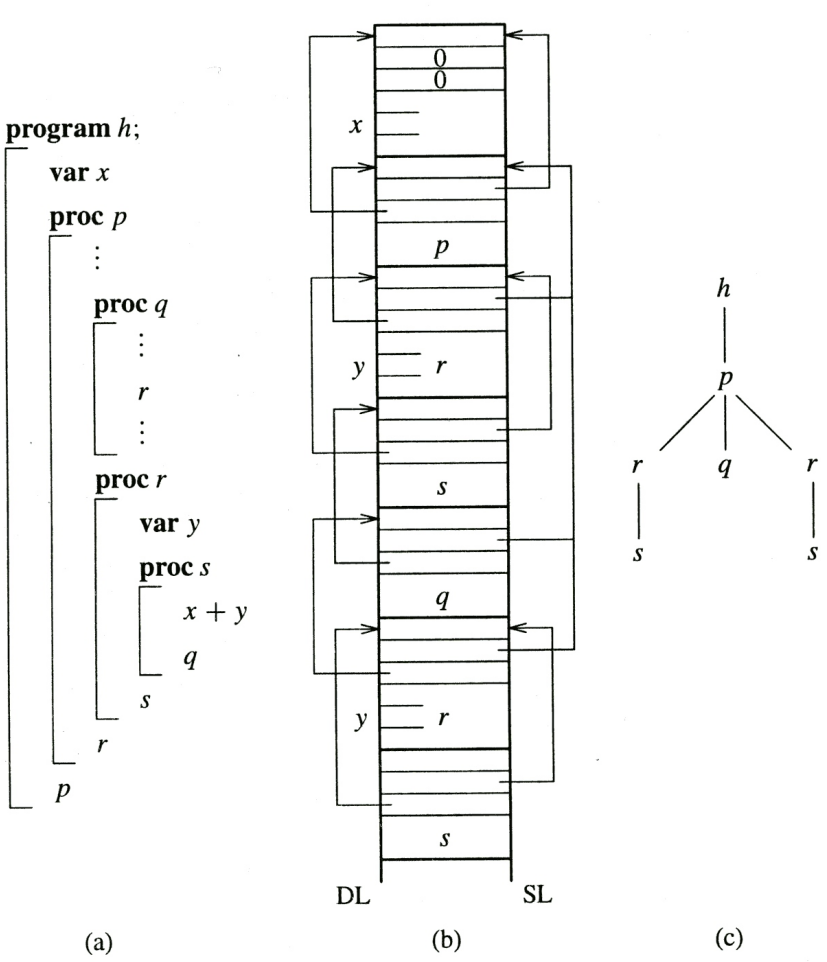

Figura 6.10: Pila de marcos con enlaces estáticos y dinámicos

Defining occurrence		Applied occurrence	
p	1	p	1
q	2	q	4
r	2	r (in p)	2
s	3	r (in q)	3
		s	3

Figura 6.11: Apariciones de definición y de uso con profundidades de anidamiento

La diferencia $m - n$ de profundidades de anidamiento nos dice cuántos enlaces estáticos hay que remontar en la pila de marcos. Si $m - n = 0$, la variable es local y su dirección absoluta es $dir(x) = \texttt{MP} + \rho(x)$. Si MP es el registro de la unidad en curso, el cálculo a realizar en general es $dir(x) = base(m - n, \texttt{MP}) + \rho(x)$, donde:

$$\begin{cases} base(0, \texttt{MP}) & = \texttt{MP} \\ base(d+1, \texttt{MP}) & = base(d, \texttt{STORE}[\texttt{MP} + 1]) \end{cases}$$

⋆ Las siguientes instrucciones-P dan soporte a la carga y almacenamiento a/desde la cima de la pila de variables posiblemente no locales. Las cantidades estáticas p y q representan respectivamente la diferencia de profundidades de anidamiento y la dirección relativa de la variable en su marco.

lod $p\ q$	$\equiv$	$\texttt{SP} := \texttt{SP} + 1;$	Apila contenido
		$\texttt{STORE}[\texttt{SP}] := \texttt{STORE}[base(p, \texttt{MP}) + q]$	
lda $p\ q$	$\equiv$	$\texttt{SP} := \texttt{SP} + 1;$	Apila dirección
		$\texttt{STORE}[\texttt{SP}] := base(p, \texttt{MP}) + q$	
str $p\ q$	$\equiv$	$\texttt{STORE}[base(p, \texttt{MP}) + q] := \texttt{STORE}[\texttt{SP}];$	Almacena y desapila contenido
		$\texttt{SP} := \texttt{SP} - 1$	

⋆ El esquema $code_L$ presentado en la Sección 6.4.5 ha de modificarse consecuentemente para direccionar las variables relativas a su marco, en vez con direcciones absolutas como hasta ahora. Basta sustituir

$$code_L\ (x\ r)\ \rho = \textbf{ldc}\ \rho(x);\ code_M(r)\ \rho$$

por

$$code_L\ (x\ r)\ \rho = \textbf{lda}\ (pa_{uso}(x) - pa_{def}(x))\ \rho(x);\ code_M(r)\ \rho$$

6.5.2. Secuencia de llamada

⋆ Supongamos una llamada $p(e_1, \ldots, e_k)$ desde un cierto procedimiento q, donde $e_1, \ldots, e_k$ son las expresiones de los parámetros reales, y supongamos conocidos por el compilador:

- $m = pa_{uso}(p)$. Como se ha dicho, es igual a $pa_{def}(q) + 1$.
- $n = pa_{def}(p)$.
- $s =$ longitud de la zona de parámetros formales de p.
- $l =$ dirección de comienzo del código de p.

⋆ En primer lugar definimos el esquema $code_A$, que generaliza el paso de un parámetro. El esquema apila el valor o la dirección del parámetro según el mecanismo de paso:

$$code_A\ e_j\ \rho = \begin{cases} code_R\ e_j\ \rho & \text{, si } e_j \text{ de tipo simple pasado por valor} \\ code_L\ x\ \rho; \\ \textbf{movs}\ size(t) & \text{, si } e_j = x \text{ de tipo no simple } t \text{ pasado por valor} \\ code_L\ x\ \rho & \text{, si } e_j = x \text{ expresión izquierda pasada por referencia} \end{cases}$$

⋆ Donde la instrucción **movs** copia un bloque de memoria:

$$\begin{array}{lcl} \textbf{movs}\ q & \equiv & \textbf{for}\ i := q-1\ \textbf{down to}\ 0\ \textbf{do} \\ & & \quad \texttt{STORE}[\texttt{SP}+i] := \texttt{STORE}[\texttt{STORE}[\texttt{SP}]+i]; \\ & & \texttt{SP} := \texttt{SP}+q-1 \end{array}$$

⋆ La secuencia completa de llamada es:

$$\begin{array}{lcl} code\ (p(e_1, \ldots, e_k))\ \rho & = & \textbf{mst}\ (m-n); \\ & & code_A\ e_1\ \rho; \\ & & \ldots \\ & & code_A\ e_k\ \rho; \\ & & \textbf{cup}\ s\ l \end{array}$$

⋆ La instrucción **mst** se encarga de establecer los enlaces estático y dinámico, salvar el registro `EP` del llamante y dejar `SP` en situación de comenzar el paso de parámetros:

$$\begin{array}{lcl} \textbf{mst}\ p & \equiv & \texttt{STORE}[\texttt{SP}+2] := base(p, \texttt{MP}); \\ & & \texttt{STORE}[\texttt{SP}+3] := \texttt{MP}; \\ & & \texttt{STORE}[\texttt{SP}+4] := \texttt{EP}; \\ & & \texttt{SP} := \texttt{SP}+5 \end{array}$$

⋆ La instrucción **cup** establece el nuevo valor de `MP`, salva la dirección de retorno y salta al procedimiento llamado:

$$\begin{array}{lcl} \textbf{cup}\ p\ q & \equiv & \texttt{MP} := \texttt{SP} - (p+4); \\ & & \texttt{STORE}[\texttt{MP}+4] := \texttt{PC}; \\ & & \texttt{PC} := q \end{array}$$

6.5.3. Secuencia de entrada, ejecución y terminación

⋆ Una declaración de procedimiento responde al formato:

```
procedure p (parametros formales);
  declaraciones de procedimientos anidados;
  declaraciones de variables locales;
  cuerpo ejecutable
end
```

⋆ Suponemos conocidos por el compilador los siguientes datos estáticos:

- l_e = longitud de la parte estática del marco de activación. Esta incluye la zona organizativa, la de parámetros fomales y la de variables locales.
- l_v = longitud de la pila de evaluación de expresiones.
- d_p = dirección donde comienza el código del cuerpo del procedimiento p.
- ρ_p = entorno de direcciones con el que se compila el cuerpo de p.

⋆ La disposición completa del código del procedimiento p es:

$$
\begin{array}{rl}
 & \textbf{ssp } l_e; \\
 & \textit{rellenar los descriptores de las posibles matrices dinámicas} \\
 & \textit{crear las matrices dinámicas y actualizar } \texttt{SP} \\
 & \textbf{sep } l_v \\
 & \textbf{ujp } d_p \\
 & \textit{código de los procedimientos anidados} \\
d_p: & \textit{code } (\textit{cuerpo}_p)\ \rho_p \\
 & \textbf{retp}
\end{array}
$$

⋆ La instrucción **ssp** actualiza `SP` al final de la parte estática:

$$\textbf{ssp } q \ \equiv\ \texttt{SP} := \texttt{MP} + q - 1$$

⋆ La instrucción **sep** establece el registro `EP`, y comprueba si hay colisión entre la pila y el montón:

$$
\begin{array}{rcl}
\textbf{sep } q & \equiv & \texttt{EP} := \texttt{SP} + q; \\
 & & \text{si } \texttt{EP} \geq \texttt{NP} \\
 & & \text{entonces } \textit{error "memory overflow"}
\end{array}
$$

⋆ La intrucción **retp** ejecuta la **secuencia de terminación** de un procedimiento: desapila el marco de activación, restaura el `EP` del llamante, vuelve a realizar el test de colisión (durante la ejecución de p ha podido modificarse `NP` a la baja), y finalmente salta a la dirección de retorno:

$$
\begin{array}{lcl}
\textbf{retp} & \equiv & \texttt{SP} := \texttt{MP} - 1; \\
 & & \texttt{EP} := \texttt{STORE}[\texttt{MP} + 3]; \\
 & & \text{si } \texttt{EP} \geq \texttt{NP} \\
 & & \text{entonces } \textit{error "memory overflow"} \\
 & & \text{si no} \\
 & & \texttt{PC} := \texttt{STORE}[\texttt{MP} + 4]; \\
 & & \texttt{MP} := \texttt{STORE}[\texttt{MP} + 2]
\end{array}
$$

- Si p es una función, se supone que su ejecución ha dejado en $\texttt{STORE}[\texttt{MP}]$ el resultado. La secuencia de terminación ejecuta **retf** en lugar de **retp**, cuya semántica es idéntica excepto por el hecho de que ejecuta $\texttt{SP} := \texttt{MP}$ en lugar de $\texttt{SP} := \texttt{MP} - 1$. De este modo, la pila del llamante queda incrementada en uno, y en la nueva cima se encuentra el resultado de la función.

- El esquema de $code_L$ presentado al final de la Sección 6.5.1 es apropiado para acceder dentro del código de p tanto a sus variables locales como a los parámetros pasados por valor, que a todos los efectos son tratados por el lenguaje como si fueran variables locales.

- En cambio no es apropiado para acceder a los parámetros pasados por referencia, ya que lo que se ha almacenado en el marco de activación es la **dirección** de los mismos. Si x es un parámetro formal pasado por referencia, $\rho(x)$ nos da la dirección relativa del lugar en el marco donde se encuentra la dirección del parámetro real que corresponde a x. Para obtener la dirección del parámetro real, debemos modificar de nuevo el esquema $code_L$:

  $$code_L\ (x\ r)\ \rho = \textbf{lod}\ 0\ \rho(x);\ code_M\ (r)\ \rho$$

  La instrucción **lod** carga el contenido de $\texttt{MP}+\rho(x)$, es decir la dirección del parámetro real.

Notas bibliográficas

Este tema está basado y se trata más ampliamente en (**?**, Capítulo 2), cuya lectura se recomienda como complemento.

Ejercicios

1. Dar un algoritmo para calcular, a partir del árbol abstracto de una expresión, la longitud de la pila de evaluación de expresiones requerida por la máquina-P.

2. Compilar, indicando los pasos intermedios seguidos, y suponiendo que $\rho(a) = 5, \rho(b) = 6, \rho(c) = 7$, el siguiente fragmento,