7.4. UTILIDADES DE LAS PILAS

- Llamadas a subprogramas
- Paso de programas recursivos a iterativos
    - Un caso especial, quick_sort iterativo.
- Equilibrado de símbolos
- Tratamiento de expresiones aritméticas
    - Evaluación de expresiones postfijas
    - Conversión de expresiones infija → postfija

7.4.1. Llamadas a subprogramas

Cada vez que se llama a un subprograma (subrutina o función) es necesario guardar el estado del programa que realiza la llamada:

- Dirección del programa a la que debe volver cuando termine,
- las variables del programa, junto con sus valores antes de la llamada.

El entorno del programa se guarda en una pila.
Esta operación será especialmente importante cuando se realicen llamadas recursivas. En estos casos, si no se realiza un control del tamaño de la pila puede producirse un desbordamiento.

7.4.2. Paso de programas recursivos a iterativos

Como ya hemos visto en el Tema 4, es una técnica de programación relativamente simple, pero:

- Puede ser computacionalmente costosa (en términos de consumo de memoria),
- existen lenguajes que no admiten recursividad (y que, por lo tanto, es necesario simular)
  - Llamar continuamente a subprogramas implica usar continuamente la memoria del sistema (organizada como pila) y los recursos del sistema operativo.
  - Simular procedimientos recursivos implica usar pilas (para guardar, en la simulación de cada llamada recursiva, el estado actual del problema).
  - El uso de pilas evita el uso de etiquetas junto con GOTO para eliminar la recursividad.

Ejemplo de Quick Sort Iterativo utilizando Pilas

Quick_sort() es un algoritmo cuya definición es recursiva. Para simularlo de forma iterativa sólo tendremos que almacenar el entorno del problema en cada llamada recursiva. Para ello se recurre a dos pilas: PILASUP y PILAINF, que almacenan los límites superior e inferior del vector que estamos ordenando.

Inicialmente:

En cada momento, los límites [inferior..superior] de la porción de vector por ordenar están en el tope de las pilas PILAINF y PILASUP:

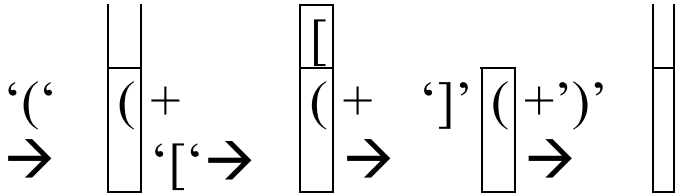
Algoritmo QS_iterativo (V, N) es
V: Vector [1..N] de numérico;
N: Numérico;
inf, sup: numérico;
PilaInf, PilaSup: PILA;
INICIO
    inf := 1;
    sup := N;
    {Donde vamos a guardar los extremos inferior y superior del vector, sobre los que haremos la primera partición}
    Crear_pila (PilaInf);
    Crear_pila (PilaSup);
    Push (PilaInf, inf);
    Push (PilaSup, sup);
    ReduceIterativo (V, N, PilaInf, PilaSup);
FIN

Algoritmo ReduceIterativo (V, N, PI, PS) es
V: Vector [1..N] de numérico;
N: Numérico;
PI, PS: PILA; {Nombres locales de las pilas PILAINF y PILASUP}
inf, sup, izd, der: numérico;
pivote: numérico;
resp: lógico;
INICIO
{PI y PS se van a ir llenando y vaciando a la vez. Cuando ambas estén vacías, significa que ya no hay partes del vector a ordenar.}
vacía?(PI, resp);
mientras not resp **hacer**
    {Obtiene los extremos [inf..sup] del vector; equivale a la entrada de la función recursiva}
    pop(PI, inf);
    pop(PS, sup);
    pivote := V((inf + sup) div 2);
    izd := inf;
    der := sup;
    {realizar la partición}

```
    repetir
        mientras (v(izd) < pivote) hacer  izd := izd + 1; fin mientras;
        mientras (v(der) > pivote) hacer  der := der - 1; fin mientras;
        si izd <= der entonces
            intercambia (V(izd), V(der));
            izd := izd + 1;
            der := der - 1;
        fin si;
    hasta que (izd > der); {realiza la partición del vector [inf..sup] ➔ [inf..der] y [izd..sup]}
    si inf < der entonces {equivalente a llamar en la versión recursiva: sort (V, inf, der);}
        push(PI, inf);
        push(PS, der);
    fin si;
    si izd < sup entonces {equivalente a llamar en la versión recursiva: sort (V, izd, sup);}
        push(PI, izd);
        push(PS, sup);
    fin si;
    vacía? (PI, resp);
fin mientras; {¿existen porciones para ordenar?}
FIN
```

7.4.3. Equilibrado de símbolos

- Utilizado por los compiladores para comprobar la sintaxis: por cada elemento clave de apertura tiene que haber uno de cierre. Ejemplo: (), [], {}, begin-end.
- En una pila se guardan los elementos (palabras o símbolos clave). Si el elemento leído se cancela con el del tope de la pila se elimina (*pop()*). Si no, se mete en la pila (*push()*).
- Si al finalizar el análisis la pila está vacía, esa sintaxis es correcta.



Algoritmo Sintaxis (fich: cadena, ok: lógico) es
f: fichero de ELTO;
pal: ELTO;
P: PILA;
Eof: lógico;
INICIO
    Abrir_f (f, fich, "l");
    crear_pila (P, ok);
    Fin_f (f, eof);
    mientras not eof **hacer** {recorrido secuencial de un fichero}
        Leer_palabra (f, pal); {Leer palabra es más complejo que leer_f (f, V)}
        cancelar (P, pal);
        fin_f (f, eof);
    fin mientras;
    Cerrar_f (f);
    vacía? (P, ok);
    si ok **entonces**
        Escribir "Sintaxis Correcta";
    finsi
FIN

```
Algoritmo Cancelar (P,x) es
P,: PILA;
x, T: ELTO;
resp: lógico;
INICIO
    vacía? (P, resp);
    si resp entonces
        push (P, x);
    sino
        tope(P, T);
        se_cancelan? (T, x, resp);
        si resp entonces
            pop(P, T);
        sino
            push(P, x);
        finsi
    finsi
FIN
```

7.4.4. Tratamiento de expresiones aritméticas

¿Cuál es el resultado de 4*5 + 6*7?

- 20 + 42 = 62, correcto
- 20 + 6 * 7= 26 * 7 = 182, error

¿A qué se debe el error? Se han utilizado mal las precedencias.
¿Cómo puede solucionarse?

- Paréntesis: (4*5) + (6 * 7)
- Usando otra notación para los símbolos.
  Tipos de notación :
  - Infija: A ***op*** B ➔ 4 * 5 + 6 * 7
  - Postfija: A B ***op*** ➔ 4 5 * 6 7 * +
  - Prefija: ***op*** A B ➔ + * 4 5 * 6 7

Una ventaja de la notación postfija es que no necesita paréntesis.

7.4.4.1. Evaluación de notación postfija mediante una pila

¿Cómo usar una pila?

1. Se lee la entrada, elemento a elemento, x:

   Si es un operando → se mete en la pila: *push(P, x)*

   Si es un operador → se sacan los operandos, se realiza la operación y se guarda el resultado en la pila:

   pop (P, op1);

   pop(P, op2);

   evaluar (x, op1, op2, res);

   push(P, res)

2. Al terminar de leer la entrada, en el tope de la pila P debe quedar el resultado de la operación

Entrada	Acción	Estado de la Pila
4 5 * 6 7 * +	*Push()*	4
5 * 6 7 * +	*Push()*	5 4
* 6 7 * +	*Evaluar + push()*	20
6 7 * +	*Push()*	6 20
7 * +	*Push()*	7 6 20
* +	*Evaluar + push()*	42 20
+	*Evaluar + push()*	62

{Versión sin comprobaciones de error}
Algoritmo Evaluar_post (R, fich) es
R: ELTO; {Resultado}
fich: cadena; {Nombre del fichero donde se encuentra la expresión}
P: PILA; {Utilizada como almacenamiento intermedio}
op1, op2, T: ELTO;
eof, resp, ok: lógico;
f: fichero de ELTO;
INICIO
    Abrir_f (f, fich, "l"); {la entrada se gestiona como un fichero}
    Crear_pila (P, ok);
    Fin_f (f, eof);
    mientras not eof **hacer** {recorrido secuencial de la entrada}
        Leer_f (f, T);
        operando? (T, resp);
        si not resp entonces {es un operador, hay que sacar los operandos }
            pop (P, op2);
            pop (P, op1);
            evaluar (T, op1, op2, R);
            push (P, R);
        sino {es un operando}

```
                *push* (P, T);
            finsi;
            Fin_f (f, eof);
        fin mientras
        Cerrar_f (f);
        *Tope* (P, R);
FIN

{Versión con comprobaciones de error}
Algoritmo Evaluar_post_2 (R, fich) es
R: ELTO; {Resultado}
fich: cadena; {Nombre del fichero donde se encuentra la expresión}
P: PILA; {Utilizada como almacenamiento intermedio}
op1, op2, T: ELTO;
eof, resp, ok, error: lógico;
f: fichero de ELTO;
INICIO
        Abrir_f (f, fich, "l");
        *Crear_pila* (P, ok);
        error := falso;
        Fin_f (f, eof);
```

```
mientras not eof and not error hacer
        Leer_f (f, T);
        operando? (T, resp);
        si not resp entonces {es un operador, hay que sacar los operandos}
                pop (P, op2);
                operando? (op2, resp);
                si resp = cierto entonces {es el segundo operando}
                        pop (P, op1);
                        operando? (op1, resp);
                        si resp = cierto entonces {es el primer operando}
                                evaluar (T, op1, op2, R);
                                push (P, R);
                        sino error:=cierto;
                        finsi
                sino error := cierto;
                finsi
        sino {es un operando}
                push (P, T);
        finsi;
        Fin_f (f, eof);
fin mientras
```

```
    Cerrar_f (f);
    Tope (P, R);
FIN
```

Resumen

- No es necesario saber la precedencia de los operadores (si la expresión está bien construida),
- el resultado es un procedimiento lineal (en el tamaño de la entrada)

Ejercicio: Solucionar este problema para operadores unarios

7.4.4.2. Conversión de notación infija a postfija

El caso anterior resulta de utilidad si se dispone de una operación que obtenga la notación postfija a partir de una operación infija.

Ejemplo: a + b * c + (d * e + f) * g ➔ a b c * + d e * f + g * +

Usando una pila este proceso puede realizarse de la siguiente forma:

1. Leyendo uno por uno los elementos de la entrada y si
   a) es un operando, se manda a la salida;
   b) es un "(" → *push* (P, "(")
   c) es un ")" → hacer *pop* (P, t) hasta que se encuentre ")"
   d) es un operador *op*, entonces
      d.1) sacar de la pila, *pop* (P, t), los operadores con prioridad **mayor o igual** a *op*: de menor a mayor:
         - (
         - +,-
         - *, /
      d.2) *push* (P, *op*)
2. Cuando se llega al final de la entrada, se vacía la pila P sobre la salida.

Ejemplo: Entrada = a + b * c + (d * e + f) * g
En el siguiente ejemplo el tope de la pila es el elemento más a la izda.

Entrada	**Operación**	**PILA**	**Salida**
‘a’	a la salida		a
‘+’	*Push*	‘+’	a
‘b’	a la salida	‘+’	a b
‘*’	→ ‘+’ < ‘*’ → *push*	‘*’ ‘+’	a b
‘c’	a la salida	‘*’ ‘+’	a b c
‘+’	mientras precedencia tope >= precedencia operador hacer *pop* ‘*’ ≥ ‘+’ → *pop* ‘+’ ≥ ‘+’ → *pop*	‘+’	a b c * +
‘(‘	→ *push*	‘(‘ ‘+’	a b c * +
‘d’	a la salida	‘(‘ ‘+’	a b c * + d

‘*’	→ ‘(‘ < ‘*’	‘*’ ‘(‘ ‘+’	a b c * + d
‘e’	a la salida	‘*’ ‘(‘ ‘+’	a b c * + d e
‘+’	→ ‘*’ ≥ ‘+’ → *pop* ‘*’ y *push* ‘+’	‘+’ ‘(‘ ‘+’	a b c * + d e *
‘f’	a la salida	‘+’ ‘(‘ ‘+’	a b c * + d e * f
‘)’	→ repetir *pop* hasta encontrar ‘(‘	‘+’	a b c * + d e * f +
‘*’	→ ‘+’ < ‘*’	‘*’ ‘+’	a b c * + d e * f +
‘g’	a la salida	‘*’ ‘+’	a b c * + d e * f + g
EOF	vaciar pila		a b c * + d e * f + g * +

Algoritmo conversión_infija_postfija (infija, postfija) es
infija, postfija: cadena; {nombre de los ficheros}
inf, post: fichero de carácter;
c, top: carácter;
prec1, prec2: entero;
P: pila;
eof, resp: lógico;
INICIO
Abrir_f (inf, infija, “l”);
Abrir_f (post, postfija, “e”);
Fin_f (inf, eof);
mientras not eof hacer {Leer expresión infija}
    Leer_f(inf, c);
    es_operando? (c, resp);
    si resp = cierto entonces {Se manda directamente a la salida}
        Escribir_f (post, c);
    sino {es un operador}
        según (c) hacer
            ‘(‘:   *push*(P, c);

```
')':  repetir
            pop (P, top);
            si top <> '(' entonces Escribir_f (post, top);
            finsi;
      hasta que top = '(';
otros:  {operador distinto de los paréntesis}
      {Sacar operadores con mayor precedencia}
      vacía? (P, resp);
      si resp = falso entonces
            tope (P, top);
            precedencia(c, prec1);
            precedencia(tope, prec2);
            mientras prec2 >= prec1 AND not resp hacer
                  pop(P, top);
                  Escribir_f (post, top);
                  vacía? (P, resp);
                  si resp = falso entonces
                        tope(P, top);
                        predecendia(top, prec2);
```

```
                        finsi
                    fin mientras;
                finsi;
                push (P, c); {Meto en la pila el operador que había leído}
            fin según;
        finsi
        Fin_f (inf, eof);
fin mientras;
Cerrar_f (inf);  {Se ha terminado la entrada} {Resta vaciar la pila de operadores}
vacía? (P, resp);
mientras not resp hacer
        pop(P, top);
        Escribir_f (post, top);
        vacía? (P, resp);
fin mientras;
Cerrar_f(post);
FIN
```