Ejemplos de aplicación del TAD pila

1. Evaluación de expresiones postfijas

Una expresión aritmética en **notación postfija** (o notación polaca inversa) es una secuencia formada por símbolos de dos tipos diferentes: **operadores** (para simplificar, consideraremos únicamente los operadores aritméticos binarios +, -, * y /) y **operandos** (para simplificar pensaremos en identificadores de una sola letra). Cada **operador se escribe detrás** de sus operandos.

Por ejemplo, a la expresión siguiente, escrita en la notación habitual (**infija**):

`a*b/c`

le corresponde la siguiente expresión en notación postfija:

`ab*c/`

Si en la expresión infija aparecen **paréntesis**, estos cambian la correspondiente expresión postfija sólo si los paréntesis alteran el orden de prioridad de los operadores:

`a/b+c*d-e*f`, traducido a notación postfija es: `ab/cd*+ef*-`

`a/(b+c)*(d-e)*f`, se traduce en cambio por: `abc+/de-*f*`

Tres **ventajas** importantes de la notación postfija frente a la convencional infija son las siguientes:

- En notación postfija **nunca son necesarios los paréntesis**.
- En notación postfija **no es necesario definir prioridades** entre operadores.
- Una expresión postfija puede **evaluarse de forma muy sencilla**, como veremos enseguida.

En el siguiente apartado de la lección veremos un algoritmo para traducir expresiones de notación infija a notación postfija. Veamos ahora cómo evaluar una expresión escrita en notación postfija.

Una expresión en notación postfija puede ser **evaluada** haciendo un **recorrido de izquierda a derecha**. Cuando se encuentra un operando, se apila en una **pila de operandos**. Cuando se encuentra un operador, se desapilan dos operandos de la pila, se realiza la correspondiente operación y el resultado se apila en la pila de operandos. Este método de evaluación es mucho más sencillo que el proceso necesario para evaluar una expresión escrita en notación infija.

A continuación se presenta el algoritmo de evaluación de expresiones en notación postfija:

```
función evaluar(e:expresión) devuelve real
{ Devuelve el valor real de la expresión postfija e. }
importa pilasDeSímbolos
variables s,o1,o2,r:símbolo; p:pilaDeSímbolos
principio
  creaVacía(p);
  s:=siguienteSímbolo(e);
  mientrasQue s≠final hacer
    si esOperando(s)
    entonces
      apilar(p,s)
    sino
      o2:=cima(p);
      desapilar(p);
      o1:=cima(p);
      desapilar(p);
      r:=operar(o1,o2,s);
      apilar(p,r)
    fsi;
    s:=siguienteSímbolo(e)
  fmq;
  devuelve(cima(p));
  desapilar(p)
fin
```

Se ha supuesto lo siguiente:

- Se usa el módulo `pilasDeSímbolos`, en el cual se ha definido el tipo `símbolo`, cuyos valores pueden ser operandos, operadores o el valor especial `final`; además, existe la función `esOperando` que devuelve verdad si y sólo si el símbolo enviado como argumento es un operando; existe también

la función `operar`, que a partir de dos operandos y un operador devuelve un nuevo operando que es el resultado de realizar la operación.

- El TAD genérico `pila` se supone particularizado para datos de tipo `símbolo` en el módulo `pilasDeSímbolos`, es decir, el parámetro formal `elemento` de la especificación del TAD `pila` debe sustituirse por el tipo `símbolo`.
- Se supone predefinido el tipo `expresión` como una secuencia de `símbolos`. Además, existe la función `siguienteSímbolo` que devuelve el siguiente `símbolo` de una `expresión`.

2. Traducción de expresiones infijas a postfijas

Una vez resuelto el problema de evaluar una expresión escrita en notación postfija, podemos plantearnos el de realizar la traducción de expresiones infijas a postfijas para, así, tener resuelto el de evaluar expresiones infijas.

Comparando una expresión infija con su correspondiente postfija, puede verse en primer lugar que los **operandos mantienen el mismo orden** en ambas notaciones. Por tanto, únicamente hay que "mover" operadores y quitar paréntesis (si existen).

Un primer algoritmo de traducción es el siguiente:

- Añadir a la expresión un par de paréntesis por cada operador. Esto significa añadir paréntesis redundantes con las reglas de prioridad. Por ejemplo, de la expresión:
  `a/b+c*d-e*f`
  pasar a la expresión:
  `(((a/b)+(c*d))-(e*f))`
- Mover todos los operadores de forma que sustituyan a sus correspondientes paréntesis derechos. En el ejemplo anterior:
  `(((ab/(cd*+(ef*-`
- Borrar todos los paréntesis izquierdos. Es decir:
  `ab/cd*+ef*-`

El **problema** del algoritmo anterior es que **requiere dos recorridos** de la expresión para traducirla: el primero para añadir todos los paréntesis redundantes y el segundo para mover los operadores y eliminar paréntesis.

La solución a ese problema se obtiene de la siguiente forma: puesto que el orden de los operandos es el mismo, cada vez que es leido un operando, se escribe en el resultado; cada vez que se lee un operador, hay que decidir si debe escribirse ya en el resultado o almacenarse en un dato auxiliar (veremos que será una pila) hasta el momento de su escritura.

Por ejemplo, para traducir la expresión `a+b*c` a su correspondiente postfija `abc*+`, hay que realizar los siguientes pasos:

siguiente símbolo	pila	resultado	comentario
	vacía		inicialización de la pila
`a`	vacía	`a`	el operando se escribe ya
`+`	`+`	`a`	el operador se apila porque no hay otro
`b`	`+`	`ab`	el operando se escribe ya
`*`	`+*`	`ab`	`*` tiene más prioridad que `+`, luego se apila
`c`	`+*`	`abc`	el operando se escribe ya
final	vacía	`abc*+`	se vuelca la pila sobre el resultado

Veamos ahora un ejemplo con paréntesis: la expresión `a*(b+c)/d`.

siguiente símbolo	pila	resultado	comentario
	vacía		inicialización de la pila
a	vacía	a	el operando se escribe ya
*	*	a	el operador se apila porque no hay otro
(	*(	a	el paréntesis izquierdo se apila siempre
b	*(	ab	el operando se escribe ya
+	*(+	ab	a un paréntesis izquierdo sólo lo saca
			el derecho
c	*(+	abc	el operando se escribe ya
)	*	abc+	se desapila hasta el paréntesis izquierdo,
			escribiendo el + que se ha desapilado
/	/	abc+*	/ tiene igual prioridad que *, luego se
			saca * de la pila y se mete /
d	/	abc+*d	el operando se escribe ya
final	vacía	abc+*d/	se vuelca la pila sobre el resultado

En general, el problema se resuelve de la siguiente forma:

- a cada operador y a los paréntesis se les asigna una “prioridad en pila” para cuando están dentro de la pila y otra “prioridad de llegada” para cuando son leidos de la entrada;
- cada vez que se lee un operador o un paréntesis izquierdo, si la pila es vacía, se apila, si no, se compara su prioridad de llegada con la prioridad en pila del símbolo que está en la cima de la pila; debe sacarse el operador de la cima de la pila cuando su prioridad en pila es mayor o igual que la prioridad de llegada del nuevo operador leido, y repetir el proceso con el nuevo operador que queda en la cima;
- cada vez que se lee un paréntesis derecho, debe desapilarse el operador de la cima, escribirse en el resultado y desapilarse también el paréntesis izquierdo que queda en la cima de la pila;
- las prioridades en pila y de llegada de los operadores +, -, * y /, y del paréntesis izquierdo son:

símbolo	prioridad en pila	prioridad de llegada
*, /	2	2
+, -	1	1
(	0	3

El algoritmo que resuelve el problema de la traducción infija a postfija es el siguiente:

```
procedimiento traducir(ent in:expresión; sal post:expresión)
{Traduce la expresión infija in a su correspondiente postfija (post).}
importa pilasDeSímbolos
variables p:pilaDeSímbolos; s:símbolo

  función prioridadDeLaPila(p:pila) devuelve 0..2
  { Pre: p es una pila de símbolos s∈{(,+,-,*,/} }
  { Post: p=pilaVacía ⇒ prioridadDeLaPila(p)=0;
          p≠pilaVacía ∧ cima(p)=( ⇒ prioridadDeLaPila(p)=0;
          p≠pilaVacía ∧ cima(p) ∈{+,-} ⇒ prioridadDeLaPila(p)=1;
          p≠pilaVacía ∧ cima(p) ∈{*,/} ⇒ prioridadDeLaPila(p)=2 }

  función prioridadDeLlegada(s:símbolo) devuelve 1..3
  { Pre: s∈{(,+,-,*,/} }
  { Post: s=( ⇒ prioridadDeLlegada(s)=3;
          (s=+)∨(s=-) ⇒ prioridadDeLlegada(s)=1;
          (s=*)∨(s=/) ⇒ prioridadDeLlegada(s)=2 }

principio
  creaVacía(p);
  iniciaExpresión(post);
  s:=siguienteSímbolo(in);
```

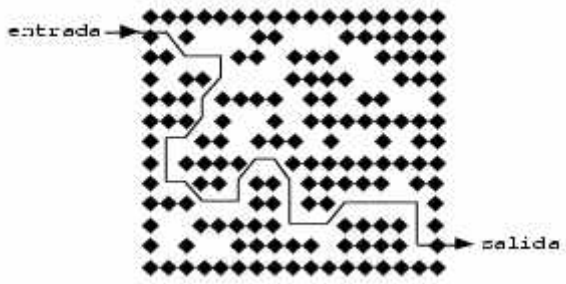
```
  mientrasQue s≠final hacer
    si esOperando(s)
    entonces
      añadeSímbolo(post,s)
    sino
      si esParéntesisDerecho(s)
      entonces
        mientrasQue not esParéntesisIzquierdo(cima(p)) hacer
          añadeSímbolo(post,cima(p));
          desapilar(p)
        fmq;
        desapilar(p) {quitar el paréntesis izquierdo}
      sino
        mientrasQue
          prioridadDeLaPila(p)≥prioridadDeLlegada(s) hacer
          añadeSímbolo(post,cima(p));
          desapilar(p)
        fmq;
        apilar(p,s)
      fsi
    fsi;
    s:=siguienteSímbolo(in)
  fmq;
  mientrasQue not esVacía(p) hacer
    añadeSímbolo(post,cima(p));
    desapilar(p)
  fmq;
  añadeSímbolo(post,final)
fin
```

Se ha supuesto lo siguiente:

- Además de la función `esOperando`, existen las funciones booleanas `esParéntesisDerecho` y `esParéntesisIzquierdo`.
- Existen los algoritmos `iniciaExpresión` y `añadeSímbolo` que crean la expresión vacía y añaden un nuevo símbolo a la derecha de una expresión, respectivamente.

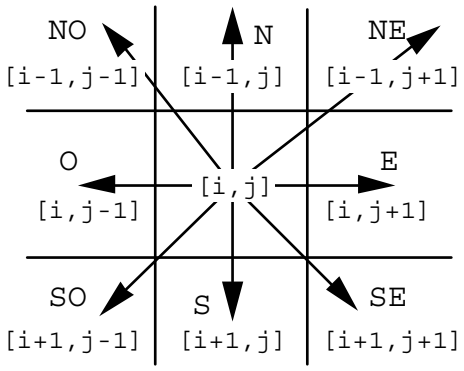
3. Recorrido de un laberinto

En este apartado va a utilizarse una pila como dato auxiliar para resolver el problema de encontrar la salida de un laberinto. Existe una entrada al laberinto y, una vez dentro de él, el paseante se encuentra con paredes que le impiden el paso en muchas direcciones. Debe elegirse un camino, en caso de que no lleve a la salida hay que volver atrás y continuar en una dirección diferente, y así hasta encontrar la única salida existente. El siguiente es un ejemplo de laberinto:



En primer lugar, para representar el laberinto podemos utilizar una estructura de datos consistente en una matriz de `m×p` componentes booleanas de forma que un valor falso en una componente indica que en la correspondiente posición del laberinto hay una pared, mientras que el valor verdad representa la existencia de un espacio libre. Supongamos que la entrada se realiza siempre por la componente `1,1` (como en la figura) y la salida por la componente `m,p`. La situación del paseante en el laberinto estará siempre descrita por la fila `i` y la columna `j` de su posición en la matriz.

Como puede verse, desde cada posición alcanzada en el interior del laberinto puede optarse por continuar en ocho direcciones diferentes (Norte, NorEste, Este, SurEste, Sur, SurOeste, Oeste y NorOeste):



No todas las posiciones `i,j` permiten moverse en ocho direcciones diferentes (si `i=1` ó `m` ó `j=1` ó `p`, entonces son menos los movimientos posibles). Para evitar ese problema, "orlaremos" la matriz que representa el laberinto con una pared (componentes con valor falso) en las filas `0` y `m+1` y en las columnas `0` y `p+1`.

La solución al problema es la siguiente: al llegar a una nueva posición se examinan todas las posibles direcciones, desde la Este a la Noreste (en el sentido de las agujas de un reloj); cada vez que se realiza un movimiento, se guarda éste en una pila (posición y dirección del movimiento); si se llega a una posición desde la que no se puede seguir, hay que volver atrás, desapilando el último movimiento y realizando el siguiente posible. Para evitar pasar dos veces por el mismo camino se necesita almacenar en otra matriz de booleanos auxiliar (inicializada con valores falso) para marcar las posiciones por las que ya se ha pasado (con valor verdad).

```
constantes m = ...; p = ...
tipo laberinto = vector[0..m+1,0..p+1] de booleano;
     dirección = (E,SE,S,SO,O,NO,N,NE);
     movimiento = registro
                    fil:1..m;
                    col:1..p;
                    dir:dirección
                  freg

procedimiento inviertePila(ent p:pilaDeMov; sal pI:pilaDeMov)
principio
  creaVacía(pI);
  mientrasQue not esVacía(p) hacer
    apilar(pI,cima(p));
    desapilar(p)
  fmq
fin

procedimiento camino(ent lab:laberinto)
{ Escribe en pantalla, si existe, un camino del laberinto que
  va de la posición 1,1 a la posición m,p. }
variables hePasado:vector[1..m,1..p] de booleano;
          mov:movimiento;
          pila,pilaInv:pilaDeMov;
          éxito:booleano;
principio
  {inicialización de la matriz de marcas a falso}
  para i:=1 hasta m hacer
    para j:=1 hasta p hacer
      hePasado[i,j]:=falso
    fpara
  fpara;
  {se parte de la casilla 1,1 hacia el Este}
  éxito:=falso;
  hePasado[1,1]:=verdad;
  mov.fil:=1;
  mov.col:=1;
  mov.dir:=E;
  creaVacía(pila);
  apilar(pila,mov);
  mientrasQue not esVacía(pila) and not éxito hacer
    {ver la posición actual y la dirección del movimiento}
    mov:=cima(pila);
    selección {cálculo de la nueva posición}
      mov.dir=N:  nuevaFil:=mov.fil-1; nuevaCol:=mov.col;
      mov.dir=NE: nuevaFil:=mov.fil-1; nuevaCol:=mov.col+1;
      mov.dir=E:  nuevaFil:=mov.fil;   nuevaCol:=mov.col+1;
      mov.dir=SE: nuevaFil:=mov.fil+1; nuevaCol:=mov.col+1;
      mov.dir=S:  nuevaFil:=mov.fil+1; nuevaCol:=mov.col;
```

```
      mov.dir=SO: nuevaFil:=mov.fil+1; nuevaCol:=mov.col-1;
      mov.dir=O:  nuevaFil:=mov.fil;   nuevaCol:=mov.col-1;
      mov.dir=NO: nuevaFil:=mov.fil-1; nuevaCol:=mov.col-1
    fselección;
    si (nuevaFil=m) and (nuevaCol=p)
    entonces  {se llega a la salida}
      mov.fil:=m;
      mov.col:=p;
      mov.dir:=E;
      apilar(mov);
      éxito:=verdad
    sino
      si lab[nuevaFil,nuevaCol] and not hePasado[nuevaFil,nuevaCol]
      entonces {nueva posición}
        hePasado[nuevaFil,nuevaCol]:=verdad;
        mov.fil:=nuevaFil;
        mov.col:=nuevaCol;
        mov.dir:=E;
        apilar(pila,mov)
      sino  {vuelta atrás}
        desapilar(pila);
        mientrasQue (mov.dir=NE) and not esVacía(pila) hacer
          mov:=cima(pila);
          desapilar(mov)
        fmq;
        si mov.dir<NE
        entonces
          mov.dir:=sucesor(mov.dir);
          apilar(p.mov)
        fsi
      fsi
    fsi
  fmq;
  si éxito
  entonces
    inviertePila(pila,pilaInv);
    mientrasQue not esVacía(pilaInv) hacer
      mov:=cima(pilaInv);
      desapilar(pilaInv);
      escribirLínea('Ir de ',mov.fil,',',mov.col,' hacia el ',mov.dir)
    fmq
  sino
    escribirLínea('¡No hay salida!')
  fsi
fin
```