Práctica 5

Implementación de estructuras lineales de datos /1

5.1 Introducción

La representación de tipos abstractos de datos es uno de los tópicos fundamentales de la algorítmica y, de un modo más general, de la informática. La representación de tipos de datos es lo que tradicionalmente se conoce con el nombre de 'Estructuras de Datos'. La importancia de la elección de la estructura de datos adecuada para representar un tipo de datos radica en el hecho de que la elección de la estructura está en relación directa con la eficacia de las operaciones, en términos de recursos de tiempo y espacio. Desde un punto de vista más general, las implementaciones se clasifican en *estáticas* y *dinámicas*.

Las implementaciones estáticas se caracterizan por tener dimensión fija a lo largo de la ejecución del programa que las utiliza. Los elementos del tipo se sitúan en una estructura de acceso directo (*tabla*). Si los elementos se colocan en posiciones contiguas de la *tabla*, la implementación se denomina *estática contigua* y si se colocan en posiciones no necesariamente contiguas, la implementación se denomina *estática enlazada*. Estas implementaciones suelen ser eficientes en tiempo de ejecución e ineficientes en espacio necesario para la ejecución, ya que al tener la estructura tamaño fijo, el espacio que utilizan es independiente del número de elementos de la estructura, sin embargo al estar implementadas en una *tabla* el acceso es muy rápido.

Las implementaciones *dinámicas* se caracterizan porque las estructuras que las soportan tienen tamaño variable durante la ejecución del programa. Las estructuras dinámicas son, por lo tanto, eficientes en espacio, pero como el acceso a los elementos de la estructura es secuencial, suelen ser más ineficaces en tiempo necesario para la ejecución. Ahora bien, es mucho más fácil cometer errores programando estructuras dinámicas que estáticas. Esto se debe a que las estructuras dinámicas se programan utilizando direcciones de memoria, lo que en general, dificulta la verificación y hace más difícil el control de los efectos laterales. Por ejemplo, pasar una dirección de memoria por valor, tiene el mismo efecto que pasar una variable por referencia y, consecuentemente, a la hora de controlar efectos laterales se debe ser muy cuidadoso, puesto que aunque no se modifique la dirección de memoria de la estructura, se puede modificar la propia estructura sin desearlo.

5.2 Pilas

Una *pila* es una organización de datos en la que el último elemento introducido en la pila es el primer elemento en salir de la pila.

5.2.1 El TAD Pila

La especificación del tipo abstracto de datos pila es la siguiente:

Tipo: tPila (elem)
Incluye: booleano
Operaciones:
nuevaPila: $\longrightarrow$ tPila
apilar: tPila, elem $\longrightarrow$ tPila
desapilar: tPila $\longrightarrow$ tPila
cima: tPila $\longrightarrow$ elem
esNula: tPila $\longrightarrow$ booleano
Ecuaciones: $\forall$e: elem; $\forall$p: tPila;
desapilar(apilar(p, e)) $\equiv$ p
cima(apilar(p, e)) $\equiv$ e
esNula(apilar(p, e)) $\equiv$ **falso**
esNula(nuevaPila) $\equiv$ **cierto**
Ecuaciones de error:
desapilar(nuevaPila) $\equiv$ **error**
cima(nuevaPila) $\equiv$ **error**

5.2.2 El módulo *pilas*

A continuación vamos a construir un módulo llamado *pilas*, que exporte el tipo abstracto de datos *tPila* visto en la sección anterior. Vamos a realizar una implementación estática contigua.

```
tipo
    tPila = tupla
        a:  tabla[1..MAX] f de tElem;
        t:  entero;
    ftupla;
ftipo
```

En esta representación los elementos de la pila se sitúan en posiciones contiguas de la tabla `a` y junto con ésta, se coloca un natural `t` que indica el número de elementos que han de ser considerados parte de la pila. Los elementos de la pila se sitúan en las primeras posiciones de la tabla `a`. Ocupa la posición `t` el último elemento introducido.

De acuerdo con esta representación, el módulo que encapsula las pilas es como sigue[1].

```
modulo pilas;
    importa
        //nombre del módulo donde se define el tipo tElem
    importa
    exporta
```

[1]Por economía de espacio se han obviado las especificaciones en el interfaz

```
        tipo
            tPila;  ftipo
        accion nuevaPila(sal p:tPila);
        accion apilar(e/s p:tPila; ent e:tElem);
        accion desapilar(e/s p:tPila);
        accion cima(ent p:tPila; sal e:tElem);
        funcion esNula(p:tPila) dev b:booleano;
    fexporta
    implementacion
        const
            MAX = 100;
        fconst
        tipo
            tPila = tupla
                a:  tabla [1..MAX] de tElem;
                t:  entero;
            ftupla;
        ftipo
        accion nuevaPila(sal p:tPila);
            p.t:= 0
        faccion;
        accion apilar(e/s p:tPila; ent e:tElem);
            p.t:= p.t +1;
            p.a[p.t]:= e
        faccion;
        accion desapilar(e/s p:tPila);
            p.t:= p.t - 1
        faccion;
        accion cima(ent p:tPila; sal e:tElem);
            e:= p.a[p.t]
        faccion;
        funcion esNula(p:tPila) dev b:booleano;
            b:= (p.t = 0);
            dev b
        ffuncion;
    fimplementacion
fmodulo
```

Se puede observar que la implementación anterior es extraordinariamente eficaz en lo referente a recursos de tiempo. Todas las operaciones tienen lugar en tiempo constante. Sin embargo, tiene dos inconvenientes en lo que respecta a la optimización del espacio: si la pila que necesitamos puede llegar a exceder en tamaño a la constante `MAX` esta implementación es insuficiente. Si se decide aumentar el tamaño de la implementación se desperdiciará mucho espacio cuando la pila no esté llena. Por otra parte, si vamos a trabajar con pilas de tamaño pequeño siempre desperdiciaremos espacio.

5.2.3 Traducción a lenguaje C

Como ya se ha visto en sesiones anteriores como se traduce un módulo de nuestro lenguaje algorítmico a un módulo en C, podría pensarse que el paso del modulo `pilas` dado en la sección anterior no debe representar ningún problema adicional a los ya vistos. No obstante, de cara a la implementación en C de estructuras de datos, es necesario hacer un par de consideraciones:

1. Nombres de las cosas: En muchas ocasiones vamos a implementar tipos de un modo genérico. Al definir el tipo `tPila` estabamos pensando en el funcionamiento genérico de las pilas. Cambiando el tipo `tElem` por cualquier tipo todo debe funcionar. Sin embargo esto no es posible en C. Suponga que crea dos módulos pila, uno para crear pilas de enteros y otro para pilas de reales. Obviamente el tipo es distinto, por lo que no pueden llamarse igual. Debemos incorporar al final del nombre el tipo del dato apilado. Nuestros tipos se van a llamar, por tanto, `PilaDeInts` y `PilaDeFloats`. Si piensa ahora en los prototipos de las funciones `desapilar` de ambos módulos, uno sería `void desapilar(PilaDeInts *)` y el otro `void desapilar(PilaDeFloats *)`. Suponga ahora un programa que necesite usar ambos módulos. El compilador se encontraría con dos declaraciones diferentes de la misma función. Para evitar esto vamos a ponerle a las acciones algorítmicas siempre un sufijo que indique el tipo al que se aplica (al menos a aquellas que se refieran a TADs que puedan contener otros tipos). Es decir, los prototipos de las funciones deben ser `void desapilarPilaDeInts(PilaDeInts *)` y el otro `void desapilarPilaDeFloats(PilaDeFloats *)`[2]

2. Cumplimineto de especificaciones. Vamos a incorporar una acción especial, denominada `error`[3] cuyo objetivo es informar de los posibles errores habidos en la utilización de la estructura. Básicamente, en la implementación de las acciones y funciones del módulo vamos a añadir la comprobación de la precondición de la acción o función y caso de que no se cumpla se hará una llamada a la acción `error` con un mensaje descriptivo del error detectado.

A modo de ejemplo veamos la implementación en C del módulo `pilaDeTElems`. El fichero `pilaDeTElems.h` sería:

```
#ifndef FFF_PILA_DE_TELEMS_H
#define FFF_PILA_DE_TELEMS_H
#include <stdbool.h> // para el tipo booleano
#include ‘‘TElem.h’’ // para el tipo tElem
#define TAMANIO_PILA_DE_TELEM 100
typedef struct pilaDeTElems {
    int cima;
    tElem valores[TAMANIO_PILA_DE_TELEM];
} PilaDeTElems;
void nuevaPilaDeTElems(PilaDeTElems *);
void apilarPilaDeTElems(PilaDeTElems *, tElem);
void desapilarPilaDeTElems(PilaDeTElems *);
void cimaPilaDeTElems(PilaDeTElems, tElem *);
bool esNulaPilaDeTElems(PilaDeTElems);
#endif
```

El fichero `pilaDeTElems.c` sería:

```
#include <stdlib.h> // para ver el prototipo exit
#include ’’pilaDeTElems.h’’ //
void errorPilaDeTElems(char s[]){
    printf(’’\n\n\nERROR en el módulo pilas:  %s \n’’, s);
    while (true)
```

[2]Hay otras soluciones más potentes, pero utilizan aspectos del lenguaje aún demasiado complejos

[3]Que también debe llevar el sufijo al que me refería antes

```
        exit(-1)
}
void nuevaPilaDeTElems(PilaDeTElems *p){
    p->cima = -1; // El indice de tablas en C es el teórico -1
}
bool llenaPilaDeTElems(PilaDeTElems p){
    return (p.cima == TAMANIO_PILA_DE_TELEM -1);
}
void apilarPilaDeTElems(PilaDeTElems *p, int x){
    if (llenaPilaDeTElems(*p)
        errorPilaDeTElems(''apilando en una pila llena.'');
    p->cima++;
    p->valores[p->cima]=x; //cuidado con el tipo tElem
}
bool esNulaPilaDeTElems (PilaDeTElems p){
    return (p.cima == -1);
}
void desapilarPilaDeTElems(PilaDeTElems *p){
    if esNulaPilaDeTElems(*p)
        errorPilaDeTElems(''desapilando en pila nula'');
    p->cima--;
}
void cimaPilaDeTElems (PilaDeTElems p, int *x){
    if esNulaPilaDeTElems(*p)
        errorPilaDeTElems(''cima en pila nula'');
    *x = p.valores[p.cima]; //cuidado con el tipo tElem
}
```

Una última precisión, si el tipo `tElem` no es un tipo elemental del lenguaje C, las asignaciones indicadas con el comentario 'cuidado con el tipo tElem' pueden representar en realidad varias asignaciones correspondientes a los distintos componentes de la estructura del tipo `tElem`.

5.3 Evaluación de expresiones

En esta sección presentamos una aplicación inmediata del concepto de pila en la implementación de un programa para evaluar expresiones aritméticas que, para simplificar el problema, supondremos sintacticamente correctas.

Una expresión aritmética, como por ejemplo $9/6 * (1 + 3) * 4 - 2 * 5$, se compone de operandos, operadores y paréntesis. En este caso, supondremos que los operandos son valores enteros y que los operadores son + (suma), - (resta), * (multiplicación) y / (división), todos ellos con operandos de tipo entero y resultado entero. Para definir el orden de evaluación de los operadores en una expresión es necesario determinar lo que se denomina *reglas de precedencia*:

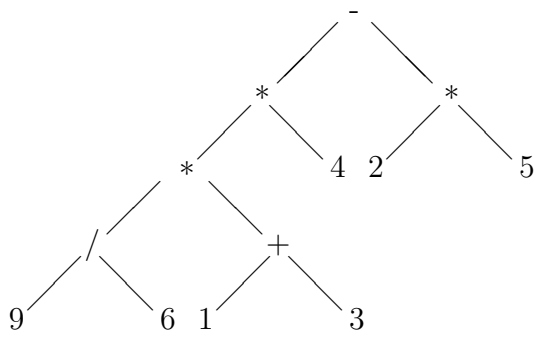
- La prioridad de los operadores. Se evalúa primero el operador de más prioridad. Suponemos que los de máxima prioridad son el producto y la división, y de menor prioridad la suma y resta.

- La asociatividad de los operadores. Suponemos que los operadores de igual prioridad, asocian por la izquierda. Es decir, se evalúa primero el operador más a la

izquierda de los de igual prioridad. Por ejemplo, en la expresión $4 * 6/2$ se realizará primero el producto de 4 y 6, y luego la división del resultado entre 2.

- Es posible cambiar la precedencia de los operadores dentro de una expresión utilizando los paréntesis: tendrán prioridad las subexpresiones contenidas entre paréntesis. Es posible cambiar el orden de evaluación de la expresión $4*6/2$ introduciendo paréntesis, por ejemplo $4 * (6/2)$ obliga a realizar la división antes que el producto.

La expresión $9/6 * (1 + 3) * 4 - 2 * 5$ dada anteriormente se evalúa en el mismo orden que la expresión $(((9/6) * (1 + 3)) * 4) - (2 * 5)$. Este orden queda correctamente expresado mediante el correspondiente *árbol sintáctico*.



Existen diversas estrategias para evaluar expresiones, aquí estudiaremos un algoritmo que se basa en el uso de pilas.

El primer inconveniente que aparece al evaluar una expresión es el tratamiento de los paréntesis, de la asociatividad y de las prioridades. Vamos a dividir el problema de la evaluación en dos subproblemas:

1. Transformar la expresión original (dada en *notación infija*) a otra notación que elimina los paréntesis y lleva implícitas la asociatividad y la prioridad de los operadores (*notación postfija*).

2. Evaluar la expresión transformada en notación postfija.

Evaluación de expresiones postfijas. Las expresiones que manipulamos habitualmente están escritas en notación infija, denominada así porque los operadores aparecen entre los operandos. En la notación postfija, los operadores aparecen después de los operandos sobre los que se aplican. Para la expresión $9/6 * (1 + 3) * 4 - 2 * 5$ su forma postfija es 9 6 / 1 3 + * 4 * 2 5 * −.

La novedad que hace interesante las expresiones postfijas es que para evaluarlas es suficiente recorrerlas de izquierda a derecha aplicando los operadores sobre los dos últimos operandos obtenidos. El algoritmo que evalúa una expresión postfija se modeliza adecuadamente utilizando una pila de enteros: se van leyendo los elementos de la expresión postfija de izquierda a derecha, si el elemento leído es un operando, se apila, si el elemento leído es un operador se extaen de la pila los dos últimos operandos introducidos, se les aplica el operador y se apila el resultado obtenido. Si la expresión postfija es correcta, al final del proceso la cima de la pila contiene un único elemento que es el resultado de la evaluación.

Ejemplo, sea la expresión 2 4 1 + * 6 3 / +, los pasos del algoritmo son

Entrada:2 4 1 + ∗ 6 3 / +	Pila:	
Entrada:+ ∗ 6 3 / +	Pila:2 4 1	
Entrada:∗ 6 3 / +	Pila:2 5	
Entrada:6 3 / +	Pila:10	
Entrada:/ +	Pila:10 6 3	
Entrada:+	Pila:10 2	
Entrada:	Pila:12	
Entrada:	Pila:	Salida:12

Transformación de notación infija a postfija. La idea general es utilizar una pila para almacenar los operandos conforme son encontrados, para más tarde desapilar estos operandos de acuerdo con su precedencia. Concretamente, la expresión en notación infija se lee también de izquierda a derecha y es procesada de acuerdo a las siguientes reglas:

- Cuando se lee un operando se lleva directamente a la salida.
- Cada vez que se lee un operador se sacan los operadores de la pila a la salida hasta encontrar un operador en la pila con menor precedencia que el operador recien leído. Entonces se apila el operador recien leído.
- Cuando se alcanza el final de la expresión infija los elementos restantes en la pila son desapilados y llevados a la salida.
- Como se pueden usar paréntesis para cambiar el orden de evaluación en las expresiones infijas, debemos incorporarlos en el proceso de conversión. Esto se consigue tratando los paréntesis como operadores que tienen una precedencia superior a la de cualquier otro operador. Además, no permitimos que los paréntesis derechos sean apilados en la pila, y sólo permitimos que un paréntesis izquierdo sea desapilado después de que haya sido leído un paréntesis derecho. Téngase en cuenta, no obstante, que los paréntesis no deben ser llevados a la salida cuando son desapilados de la pila dado que no aparecen en las expresiones postfijas.

Ejemplo, sea la expresión $2 * (4 + 1) + 6/3$, los pasos del algoritmo son

Entrada:$2 * (4 + 1) + 6/3$	Pila:	Salida:
Entrada:$*(4 + 1) + 6/3$	Pila:	Salida:2
Entrada:$(4 + 1) + 6/3$	Pila:$*$	Salida:2
Entrada:$4 + 1) + 6/3$	Pila:$*$ (	Salida:2
Entrada:$+1) + 6/3$	Pila:$*$ (	Salida:2 4
Entrada:$1) + 6/3$	Pila:$*$ (+	Salida:2 4
Entrada:$) + 6/3$	Pila:$*$ (+	Salida:2 4 1
Entrada:$+6/3$	Pila:$*$	Salida:2 4 1 +
Entrada:$+6/3$	Pila:	Salida:2 4 1 + ∗
Entrada:$6/3$	Pila:+	Salida:2 4 1 + ∗
Entrada:$/3$	Pila:+	Salida:2 4 1 + ∗ 6
Entrada:3	Pila:+ /	Salida:2 4 1 + ∗ 6
Entrada:	Pila:+ /	Salida:2 4 1 + ∗ 6 3
Entrada:	Pila:	Salida:2 4 1 + ∗ 6 3 / +

5.4 Actividad a realizar

Diseñar un programa que evalúe expresiones escritas en forma infija, con los operadores suma, resta, producto y división de números enteros. Además se prodrán utilizar paréntesis para definir el orden de las operaciones.

Se trata de dar una solución modular correcta, por lo que se propone desarrollar los módulos cuya especificación detallamos a continuación (Los tres primeros los tiene implementados en el directorio de Prácticas de miAulario).

1. Módulo `pilaDeEnteros`. Es la implementación del TAD Pila cuando los elementos a almacenar son de tipo entero. Esta pila va a ser utilizada para la evaluación de la expresión una vez en notación postfija.

```
modulo pilaDeEnteros;
    exporta
        tipo
            PilaDeEnteros;
        ftipo
        // Construye una pila p sin elementos
        accion nuevaPilaDeEnteros (sal p:PilaDeEnteros);
        // Almacena el elemento e en la pila p
        accion apilarPilaDeEnteros (e/s p:PilaDeEnteros; ent e:  entero);
        // Elimina el ultimo elemento introducido en la pila p
        accion desapilarPilaDeEnteros (e/s p:PilaDeEnteros);
        // Devuelve el ultimo elemento introducido en la pila p
        accion cimaPilaDeEnteros (ent p:PilaDeEnteros; sal x:  entero);
        // Decide si la pila p tiene elementos
        funcion esNulaPilaDeEnteros (p:PilaDeEnteros) dev b:  booleano;
    fexporta
```

2. Módulo `operadores`. Define los códigos de los operadores utilizados en las expresiones, así como la prioridad entre ellos. Los operadores para las expresiones son + (suma), - (resta), * (producto), / (división), paréntesis de apertura y paréntesis de cierre. La precedencia de los operadores según se consideren estos a la derecha o a la izquierda de otro son las que se dan en la tabla siguiente (El '$' se utiliza internamente para indicar el principio y fin de la expresión)[4]:

	+	-	*	/	(	)	$
precizqda	2	2	4	4	0	6	0
precdcha	1	1	3	3	5	0	0

```
modulo operadores;
    exporta
```

[4]Si comprueba la implementación de este módulo comprobará que una implementación, al considerar detalles del lenguaje de programación, puede modificar bastantes aspectos de la especificación. Muchas de las funciones de este módulo sólo se usan para devolver un valor constante y, por tanto, se han programado mediante macros. El fichero que le doy tambieén contiene un ejemplo de uso de una macro funcional. Aparece también el término `extern` de C. No va a ser necesario para el examen, pero si le genera curiosidad pregúntele a su profesor

```
        tipo
            Operador;
        ftipo
        // Devuelve el codigo de la operacion suma
        funcion suma() dev x:Operador;
        // Devuelve el codigo de la operacion resta
        funcion resta() dev x:Operador;
        // Devuelve el codigo de la operacion producto
        funcion producto() dev x:Operador;
        // Devuelve el codigo de la operacion division
        funcion division() dev x:Operador;
        // Devuelve el codigo del parentesis de apertura
        funcion parizqdo() dev x:Operador;
        // Devuelve el codigo del parentesis de cierre
        funcion pardcho() dev x:Operador;
        // Devuelve el codigo del delimitador de la expresion
        funcion dolar() dev x:Operador;
        // Precedencia del operador x (considerado a la izquierda de otro)
        funcion precedenciaIzquierda (x:Operador) dev n:  entero;
        // Precedencia del operador x (considerado a la derecha de otro)
        funcion precedenciaDerecha (x:Operador) dev n:  entero;
    fexporta
```

3. Módulo `símbolos`. Un símbolo es una componente simple de una expresión, puede ser un operador o un operando. Los símbolos son lo que se denomina *unidades sintácticas* de una expresión

```
modulo simbolos;
    importa
        operadores
    fimporta
    exporta
        tipo
            Simbolo;
        ftipo
        // Crea en a un simbolo operador con el operador x
        accion hazOperador (ent x:Operador; sal a:Simbolo);
        // Crea en a un simbolo operando con el valor x
        accion hazOperando (ent x:  entero; sal a:Simbolo);
        // Indica si el simbolo a corresponde a un operador
        funcion esOperador (a:Simbolo) dev b:  booleano;
        // Da en x el codigo del operador del simbolo a
        funcion operador (a:Simbolo) dev x:Operador;
        // Da en n el valor del operando del simbolo a
        funcion valor (a:Simbolo) dev n:  entero;
    fexporta
```

4. Módulo `pilaDeSimbolos`. Es la implementación del TAD Pila cuando los elementos a almacenar son de tipo `Simbolo`. Esta pila va a ser utilizada para la transformación de la expresión a notación postfija. Obviamente su interface es el que ya conoce.

5. Módulo `expresion`. Si utilizamos expresiones es lógico poder modelizarlas mediante un tipo de datos. Este módulo gestiona el almacén de la expresión postfija como una cola de un único uso[5].

[5]El paralelismo de nombres a los del TAD cola es obvio

```
modulo expresion;
    importa
        simbolos;
    fimporta
    exporta
        tipo
            Expresion;
        ftipo
        // Crea una expresion e sin elementos
        accion expresionNula (sal e:Expresion);
        // Aniade el simbolo x al final de la expresion e
        accion aniadeSimbolo (e/s e:Expresion; ent x:Simbolo);
        // Elimina el primer simbolo de la expresion e
        accion eliminaSimbolo (e/s e:Expresion);
        // Obtiene en x el primer simbolo de la expresion e
        accion primerSimbolo (ent e:Expresion; sal x:Simbolo);
        // Indica si la expresion e no tiene elementos
        accion expresionVacia (e:Expresion) dev b:  booleano;
    fexporta
```

6. Módulo `evaluador`. Añade la funcionalidad de la evaluación de expresiones en notación postfija[6]

```
modulo evaluador;
    importa
        expresion;
    fimporta
    exporta
        // Evalua la expresion en notacion postfija dada en e
        // La expresion en e debe estar bien escrita y no ser nula
        funcion evaluaPolonesa (e:  Expresion) dev v:  entero;
    fexporta
    implementación
        accion eval( e/s p:PilaDeEnteros; ent op:Operador);
            var
                v1,v2:  entero;
            fvar
            cimaPilaDeEnteros(p,v1);
            desapilarPilaDeEnteros(p);
            cimaPilaDeEnteros(p,v2);
            desapilarPilaDeEnteros(p);
            si op = suma ⟶
                v1:= v2 + v1
            [] op = resta ⟶
                v1:= v2 - v1
            [] op = producto ⟶
                v1:= v2 * v1
            [] op = division ⟶
                v1:= v2 div v1
            fsi;
            apilarPilaDeEnteros(p,v1)
```

[6]En notación de lenguaje algoritmico, cuando importamos dos modulos que exportan la misma acción/función para indicar cual se está usando ponemos el nombre del modulo seguido del nombre de la acción/función. Recuerde que esto en C no funciona. Tenemos que haber programado los módulos siguiendo la convención de nombres que le he indicado.

```
        faccion;
        funcion evaluaPolonesa(e:Expresion)dev v:  entero;
            var
                p:PilaDeEnteros; x:tSim;
            fvar
            nuevaPilaDeEnteros(p);
            mientras no expresionVacia(e) hacer
                primerSimbolo(e,x);
                si esOperador(x) ⟶
                    eval(p, operador(x))
                [] no esOperador(x) ⟶
                    apilarPilaDeEnteros(p, valor(x))
                fsi;
                eliminaSimbolo(e)
            fmientras;
            cimaPilaDeEnteros(p,v);
            dev v
        ffuncion;
    fimplementacion
fmodulo
```

7. Módulo `conversor`. Añade la funcionalidad de conversión de una expresión de notación infija a notación postfija. Nótese que estamos suponiendo que la entrada ein es una cadena de caracteres, es decir, una tabla de caracteres.

```
modulo conversor;
    importa
        expresion;
    fimporta
    exporta
        accion transformaPolonesa (ent ein:  cadena; sal eout:  expresion);
    fexporta
```

La implementación de la acción de transformación de expresiones a notación postfija puede ser la siguiente en términos de las operaciones descritas sobre los módulos anteriores:

```
accion decbin(ent s:cadena; e/s i:  entero; sal n:  entero);
    n:= 0;
    mientras s[i]≥'0' y s[i]≤'9' y i≤length(s) hacer
        n:= n*10 +(ord(s[i])-ord('0'));
        i:= i+1
    fmientras
faccion;
accion colocaSimbolo(ent y:Simbolo; e/s e:Expresion; e/s p:PilaDeSimbolos);
    var
        x:  Simbolo;
        a,b:  Operador;
    fvar
    si no esOperador(y) ⟶
        aniadeSimbolo(e, y)
    [] esOperador(y) ⟶
        cimaPilaDeSimbolos(p ,x);
        a:= operador(x);
        b:= operador(y);
```

```
        si a≠dolar o b≠dolar ⟶
            si precedenciaIzquierda(a)<precedenciaDerecha(b) ⟶
                apilarPilaDeSimbolos(p, y)
            [] precedenciaIzquierda(a)≥precedenciaDerecha(b) ⟶
                mientras precedenciaIzquierda(a)>precedenciaDerecha(b) hacer
                    aniadeSimbolo(e, x);
                    desapilarPilaDeSimbolos(p);
                    cimaPilaDeSimbolos(p,x);
                    a:= operador(x)
                fmientras;
                si precedenciaIzquierda(a)=precedenciaDerecha(b) ⟶
                    desapilarPilaDeSimbolos(p)
                [] precedenciaIzquierda(a)≠precedenciaDerecha(b) ⟶
                    apilarPilaDeSimbolos(p,y)
                fsi
            fsi
        [] a=dolar y b=dolar ⟶
            desapilarPilaDeSimbolos(p)
        fsi
    fsi;
faccion;
accion transfPolonesa( ent ein:  cadena; sal eout:  Expresion);
    var
        p:  PilaDeSimbolos;
        a:  Simbolo;
        i,x:  entero;
    fvar
    expresionNula(eout);
    nuevaPilaDeSimbolos(p);
    hazOperador(dolar,a);
    apilarPilaDeSimbolos(p,a);
    i:= 1;
    mientras i≤length(ein) hacer
        si ein[i]= ' ' ⟶
            i:= i+1
        [] ein[i]= '+' ⟶
            hazOperador(suma, a);
            colocaSimbolo(a, eout, p);
            i:= i+1
        [] ein[i]= '-' ⟶
            hazOperador(resta, a);
            colocaSimbolo(a, eout, p);
            i:= i+1
        [] ein[i]= '*' ⟶
            hazOperador(producto, a);
            colocaSimbolo(a, eout, p);
            i:= i+1
        [] ein[i]= '/' ⟶
            hazOperador(división,a);
            colocaSimbolo(a, eout, p);
            i:= i+1
        [] ein[i]= '(' ⟶
            hazOperador(parizqdo,a);
            colocaSimbolo(a, eout, p);
            i:= i+1
        [] ein[i]= ')' ⟶
```

```
                hazOperador(pardcho,a);
                colocaSimbolo(a, eout, p);
                i:= i+1
            [] ein[i]≥'0' y ein[i]≤'9' ⟶
                decbin(ein, i, x);
                hazOperando(x,a);
                colocaSimbolo(a, eout, p);
            [] ein[i]∉ {' ','+','-','*','/','(',')','0',...,'9' } ⟶
                i:= i + 1
            fsi
        fmientras
        hazOperador(dolar, a);
        colocaSimbolo(a, eout, p);
    faccion;
```

8. Programa `calculadora`. Es el programa principal que implementa la calculadora. La ejecución del programa debe seguir el esquema de todos los realizados hasta ahora: debe comenzar imprimiendo el nombre del autor, la fecha de realización y debe permitir ejecutar el programa varias veces (para distintas expresiones) hasta que el usuario decida terminar su ejecución.