APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 2: ESTRUCTURAS DE DATOS

Tipos Abstractos de Datos (TADs)

Recordemos que cuando hablamos de tipos abstractos de datos, lo esencial es que estamos pensando en caracterizar un tipo o conjunto de datos por sus propiedades y no por la manera en que puedan implementarse. Son descripciones independientes de cualquier posible implementación, de ahi el término **abstracto**.

Surgen de analizar un determinado problema para cuya resolución se pretende dar un algoritmo. En nuestro caso, nos interesa adquirir familiaridad con la notación y con ciertos tipos abstractos que ocurren con frecuencia. Plantearemos sucesivamente diferentes problemas que darán lugar a dichos TADs.

Paréntesis balanceados (TAD contador). Un problema interesante (y no del todo trivial) es el de controlar que una cierta expresión tiene balanceados sus paréntesis. Se quiere dar un algoritmo que tome una expresión (dada, por ejemplo, por un arreglo de caracteres) y devuelva verdadero si la expresión tiene sus paréntesis correctamente balanceados, y falso en caso contrario.

Este problema puede solucionarse con un algoritmo que recorre la expresión de izquierda a derecha y que utiliza un entero, que se inicializa en 0 y se incrementa cada vez que se encuentra un paréntesis que abre y se decrementa (chequeando que dicho número no sea nulo) cada vez que se encuentra un paréntesis que cierra. Al finalizar, sólo resta comprobar que dicho entero sea cero.

Esta descripción debería alcanzar para darse uno cuenta de que no es en realidad un entero lo que hace falta, sino mucho menos. Un entero es un objeto que admite todas las operaciones aritméticas, acá sólo se necesita inicializar en 0, incrementar, decrementar y controlar si su valor es o no cero. Hemos analizado qué información se necesita y cómo se la manipula. Eso da lugar al TAD contador:

TAD contador
constructores
  cero : contador
  incrementar : contador → contador
operaciones
  es_cero : contador → booleano
  decrementar : contador → contador {se aplica sólo a un contador que no sea cero}
ecuaciones
  es_cero(cero) = verdadero
  es_cero(incrementar(c)) = falso
  decrementar(incrementar(c)) = c

Los **constructores** (en este caso cero e incrementar) deben ser capaces de generar todos los valores posibles del TAD. En lo posible cada valor debe poder generarse de manera única. Intuitivamente, esto se cumple para cero e incrementar: partiendo de cero y tras sucesivos incrementos se puede alcanzar cualquier valor posible; y hay una única forma de alcanzar cada valor posible de esa manera.

Las demás **operaciones** quedan declaradas simplemente como tales. Las **operaciones** se definen con **ecuaciones** que deben cubrir todos los casos posibles, salvo los declarados que no pueden ocurrir, como en el caso de decrementar que no puede aplicarse a un contador que sea cero (a pesar de que se podría definir de manera obvia para ese caso). Más adelante veremos que a veces es necesario dar también **ecuaciones** entre **constructores**.

A continuación se presentará un algoritmo que resuelve el problema dado utilizando un contador. Asumimos para ello que el TAD contador se implementará bajo el nombre counter, que habrá un procedimiento llamado zero que implemente el constructor cero, uno llamado inc que implemente el constructor incrementar y uno llamado dec que implemente la operación decrementar. Habrá también una función is_zero que implemente la operación es_cero.[7] Vamos a utilizar informalmente la notación c $\sim$ C para indicar que c implementa C. Podemos resumir lo expresado en este párrafo de la siguiente manera:

type counter = ... {- no sabemos aún cómo se implementará -}
proc zero (**out** c: counter) {Post: c $\sim$ cero}
{Pre: c $\sim$ C} **proc** inc (**in**/**out** c: counter) {Post: c $\sim$ incrementar(C)}
{Pre: c $\sim$ C $\wedge\neg$is_zero(c)} **proc** dec (**in**/**out** c: counter) {Post: c $\sim$ decrementar(C)}
fun is_zero (c: counter) **ret** b: **bool** {Post: b = (c $\sim$ cero)}

En efecto, la primera línea indica que se define el tipo counter, aunque no precisa aún cómo. El tipo counter se utiliza para declarar las demás operaciones. El procedimiento zero declarado en la segunda línea tiene un sólo parámetro: el contador c. De su poscondición se entiende que implementa el constructor cero. El procedimiento inc declarado en la tercera línea tiene también como parámetro un contador. De su pre- y poscondición se deduce que implementa el constructor incrementar. De la misma manera puede concluirse que el procedimiento dec implementa la operación decrementar y que la función is_zero implementa la operación es_cero.

Como regla general, llamamos funciones a los comandos que devuelven un resultado sin modificar el valor de sus parámetros. En el otro extremo, llamamos procedimientos a aquellos comandos que no devuelven un resultado sino que modifican algunos de sus parámetros. Hacemos explícita en la declaración del procedimiento cuáles son los parámetros que pueden modificarse: la cláusula **in** indica que el parámetro es de entrada y por lo tanto se utiliza su valor pero no se lo modifica, la cláusula **out** indica que el mismo es de salida y por lo tanto no se utiliza su valor pero se lo modifica y la combinación de ambas (**in**/**out**) indica que es de entrada y salida, se utiliza su valor y se lo modifica.

[7]Seguiremos la convención de utilizar nombres en castellano para los constructores y operaciones especificadas por el TAD y nombres en inglés para sus implementaciones.

Asumiendo, entonces, las declaraciones anteriores, se puede chequear que los paréntesis estén bien balanceados con el siguiente algoritmo:

```
fun matching_parenthesis (a: array[1..n] of char) ret b: bool
    var i: int
    var c: counter
    b:= true
    zero(c)
    i:= 1
    {Inv: b ≡ segmento a[1,i-1] balanceado y restan cerrar paréntesis según c}
    do i ≤ n ∧ b → if a[i] = '(' → inc(c)
                      a[i] = ')' ∧ is_zero(c) → b:= false
                      a[i] = ')' ∧ ¬is_zero(c) → dec(c)
                   fi
                   i:= i+1
    od
    b:= b ∧ is_zero(c)
end fun
{Post: b ≡ segmento a[1,n] balanceado y no restan cerrar paréntesis}
```

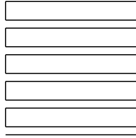
Más adelante veremos cómo implementar el TAD contador. Por ahora nos conformamos con observar que cualquier implementación completará el algoritmo que hemos dado para verificar que los paréntesis estén balanceados.

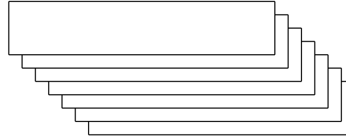
Generalización de paréntesis balanceados (TAD pila). El problema de controlar si los paréntesis están balanceados resulta menos trivial si hay diferentes tipos de paréntesis (por ejemplo, '(' y ')', '[' y ']', '{' y '}', '<', y '>', etc). Para resolverlo no alcanza con un contador, tampoco con varios contadores, ya que es necesario saber en qué orden se han ido abriendo los distintos tipos de paréntesis para controlar que se cierren exactamente en el orden inverso.

De todas formas, el algoritmo visto puede adaptarse: se recorre la expresión de izquierda a derecha y se lleva constancia de cuáles son los paréntesis que aún faltan cerrarse, y en qué orden deberían cerrarse. Dicha constancia inicialmente indica que no falta cerrar ningún paréntesis. Cada vez que se encuentra un paréntesis que abre se agrega una constancia de que dicho paréntesis debe cerrarse, y debe ser el primero en cerrarse. Cada vez que se encuentra un paréntesis que cierra debe chequearse que había paréntesis por cerrar y que el que se encontró era el primero que debía cerrarse y se debe eliminar la constancia de que falta cerrar dicho paréntesis (ya que acaba de cerrarse). Al finalizar, sólo resta comprobar que no quedan paréntesis por cerrar.

Observar que tanto cuando se agrega como cuando se quita una constancia, se trata del primer paréntesis que debe cerrarse. Estamos frente al tipo abstracto pila (stack en inglés) que se puede explicar sencillamente pensando en una pila de platos, por ejemplo que deben ser lavados. Cuando no hay platos para lavar tenemos una pila vacía, cuando se ensucia un plato se lo pone arriba de la pila, uno sólo puede ver el plato que está más alto en la pila y cuando uno va a lavar un plato saca de la pila el que está más alto (y lo lava). Uno siempre puede ver si hay platos para lavar, controlando si la pila está vacía o no.

Gráficamente puede representarse de las siguientes maneras





El gráfico de la derecha resalta el hecho de que sólo puede observarse lo que se encuentra en el primer lugar de la pila. El de la izquierda permite graficar lo que se encuentra en los otros lugares también, simplemente hay que tener presente que lo único que uno puede verdaderamente observar es lo que se encuentra en el lugar más alto.

Hemos resaltado las 5 propiedades que definen el TAD pila: la existencia de una pila vacía, la posibilidad de apilar, la de ver el último que se agregó (o sea, el primero de la pila) si la pila no está vacía, la de desapilar el último que se agregó si la pila no está vacía, la de controlar si la pila está vacía. La característica de la pila es que se extrae el último que se introdujo (LIFO = Last In, First Out).

Utilizando la notación introducida para especificar tipos abstactos de datos:

TAD pila[elem]
constructores
 vacía : pila
 apilar : elem × pila → pila
operaciones
 primero : pila → elem {se aplica sólo a una pila no vacía}
 desapilar : pila → pila {se aplica sólo a una pila no vacía}
 es_vacía : pila → booleano
ecuaciones
 primero(apilar(e,s)) = e
 desapilar(apilar(e,s)) = s
 es_vacía(vacía) = verdadero
 es_vacía(apilar(e,s)) = falso

Nuevamente, las **ecuaciones** establecen el significado de las **operaciones** primero, desapilar, es_vacía en función de los **constructores** vacía y apilar. No hay ecuaciones, en cambio, que establezcan el significado de vacía y apilar. Esto se debe a que vacía y apilar son precisamente los **constructores** del TAD pila: vacía construye la pila vacía y apilar construye pilas complejas a partir de pilas más simples. Todas las pilas se construyen usando vacía y apilar, y todas ellas son pilas diferentes entre sí.

A continuación se muestra cómo resolver el problema del balanceo de paréntesis usando una pila de caracteres. Se asumen definidas las siguientes funciones:

- match, tal que match('(') = ')', match('[') = ']', match('{') = '}', etc.
- left, tal que left('('), left('['), left('{'), etc son verdadero, en los restantes casos left devuelve falso.
- right, tal que right(')'), right(']'), right('}'), etc son verdadero, en los restantes casos, right devuelve falso.

Asumimos que se ha implementado el TAD pila (con sus nombres en inglés: stack ∼ pila, empty ∼ vacía, push ∼ apilar, top ∼ primero, pop ∼ desapilar, is_empty ∼ es_vacía). Como se hizo con el TAD contador, se asume la siguiente declaración:

```
type stack = ...                          {- no sabemos aún cómo se implementará -}
proc empty(out p:stack) {Post: p ~ vacia}
{Pre: p ~ P} proc push(in e:elem,in/out p:stack) {Post: p ~ apilar(e,P)}
{Pre: p ~ P ∧¬is_empty(p)} fun top(p:stack) ret e:elem {Post: e ~ primero(P)}
{Pre: p ~ P ∧¬is_empty(p)} proc pop(in/out p:stack) {Post: p ~ desapilar(P)}
fun is_empty(p:stack) ret b:bool {Post: b = (p ~ vacía)}
```

Asumiendo estas declaraciones, el siguiente algoritmo permite chequear que la expresión que se encuentra en el arreglo a tenga sus paréntesis correctamente balanceados:

```
fun matching_parenthesis (a: array[1..n] of char) ret b: bool
    var i: int
    var p: stack of char
    b:= true
    empty(p)
    i:= 1
    {Inv: b ≡ segmento a[1,i-1] balanceado y restan cerrar paréntesis según p}
    do i ≤ n ∧ b → if left(a[i]) → push(match(a[i]),p)
                        right(a[i]) ∧ (is_empty(p) ∨ top(p) ≠ a[i]) → b:= false
                        right(a[i]) ∧ ¬is_empty(p) ∧ top(p) = a[i] → pop(p)
                     fi
                     i:= i+1
    od
    b:= b ∧ is_empty(p)
end fun
{Post: b ≡ segmento a[1,n] balanceado y no restan cerrar paréntesis}
```

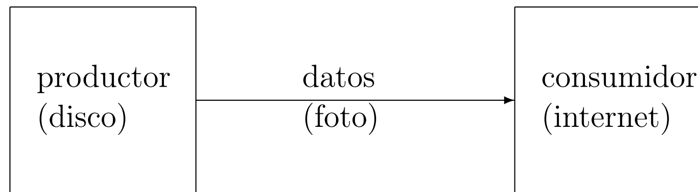
La pila es un tipo de dato muy utilizado en computación, el algoritmo que acabamos de dar para verificar paréntesis balanceados fue sólo una excusa para motivar su presentación. Se lo utiliza, por ejemplo, para implementar la recursión y para evaluar expresiones aritméticas.

Más adelante veremos cómo puede implementarse el TAD pila.

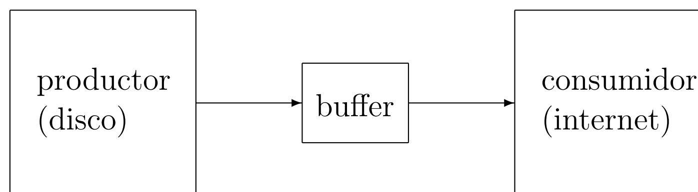
Buffer entre productor y consumidor (TAD cola). El tipo abstracto **cola** se encuentra usualmente en casi cualquier lugar donde haya, como su nombre lo indica, una cola para atención: caja de un banco, o de supermercado, etc. Un ejemplo en computación son los programas que controlan la atención de pedidos a una impresora: manejan una cola y van atendiendo los pedidos en orden de llegada. Lo mismo con las listas de espera para conseguir pasajes en un vuelo totalmente lleno. Se establece una cola de espera y a medida que se liberan asientos en ese vuelo se otorgan según el orden de llegada a la cola de espera.

El problema que utilizaremos para motivar el uso de la cola es el siguiente: imaginemos cualquier situación en que ciertos datos deben transferirse desde una unidad a otra, por ejemplo, datos (¿una foto?) que se quiere subir a algún sitio de internet desde un disco. Así, existe un agente que suministra o produce datos (el disco) y otro agente que los utiliza o consume (el sitio de internet). La velocidad en que el disco es capaz de

suministrar los datos no es la misma que la velocidad a la que los mismos pueden ser transferidos al (consumidos por el) sitio de internet.



Una manera de amortiguar la diferencia en velocidad es a través de la interposición de un buffer entre el productor y el consumidor. El mismo es capaz de alojar temporariamente los datos que el servidor va produciendo y enviárselos al consumidor a medida que los va solicitando.



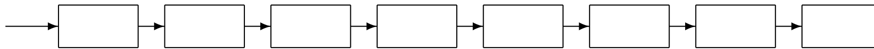
La interposición del buffer no debe afectar el orden en que los datos llegan al consumidor. El propósito es sólo permitir que el productor y el consumidor puedan funcionar cada uno a su velocidad sin necesidad de sincronización.

El buffer inicialmente está vacío. A medida que se van agregando datos suministrados por el productor, los mismos van siendo alojados en el buffer. Siempre que sea necesario enviar un dato al consumidor, habrá que comprobar que el buffer no se encuentre vacío en cuyo caso se enviará el primero que llegó al buffer y se lo eliminará del mismo.

Estas operaciones describen precisamente el tipo abstracto cola (queue en inglés), que consta de los constructores vacía (crea la cola vacía) y encolar (agrega un elemento al final de la cola) y de las operaciones primero (devuelve el primero de la cola), decolar (elimina el primero de la cola) y es_vacía (dice si la cola es vacía o no):

TAD cola[elem]
constructores
  vacía : cola
  encolar : cola × elem → cola
operaciones
  primero : cola → elem {se aplica sólo a una cola no vacía}
  decolar : cola → cola {se aplica sólo a una cola no vacía}
  es_vacía : cola → bool
ecuaciones
  primero(encolar(vacía,e)) = e
  primero(encolar(encolar(q,e’),e)) = primero(encolar(q,e’))
  decolar(encolar(vacía,e)) = vacía
  decolar(encolar(encolar(q,e’),e)) = encolar(decolar(encolar(q,e’)),e)
  es_vacía(vacía) = verdadero
  es_vacía(encolar(q,e)) = falso

Gráficamente la representación más usual es



donde a la izquierda se encuentra el primero a ser atendido, los que se agreguen lo harán por la derecha. A diferencia de la pila, la característica de la cola es que se extrae el primero que se introdujo (FIFO = First In, First Out).

Asumimos que se ha implementado el TAD cola (con sus nombres en inglés: queue ∼ cola, empty ∼ vacía, enqueue ∼ encolar, first ∼ primero, dequeue ∼ decolar, is_empty ∼ es_vacía). Se asume la siguiente declaración:

type queue = ... {- no sabemos aún cómo se implementará -}
proc empty(**out** q:queue) {Post: q ∼ vacia}
{Pre: q ∼ Q} **proc** enqueue(**in**/**out** q:queue,**in** e:elem) {Post: q ∼ encolar(Q,e)}
{Pre: q ∼ Q ∧¬is_empty(q)} **fun** first(q:queue) **ret** e:elem {Post: e ∼ primero(Q)}
{Pre: q ∼ Q ∧¬is_empty(q)} **proc** dequeue(**in**/**out** q:queue) {Post: q ∼ decolar(Q)}
fun is_empty(q:queue) **ret** b:**bool** {Post: b = (q ∼ vacía)}

Asumiendo estas declaraciones, el siguiente procedimiento implementa el buffer que resuelve el problema planteado originariamente:

```
proc buffer ()
     var d: data
     var q: queue of data
     empty(q)
     produce:= false
     demand:= false
     do forever
        if produce → receive d from producer
                     enqueue(q,d)
                     produce:= false
          demand ∧¬ is_empty(q) → d:= first(q)
                                  send d to consumer
                                  demand:= false
        fi
     od
end proc
```

En el ejemplo hemos asumido también que produce es una variable booleana compartida con el productor, que éste torna verdadera cuando produce un dato al que el buffer puede acceder gracias al comando receive. También hemos asumido que demand es una variable booleana compartida con el consumidor, que éste torna verdadera cuando espera un nuevo dato. Ni el productor ni el consumidor pueden asignar falso a dichas variables.

Como se dijo, la cola es un tipo utilizado en numerosas aplicaciones. Un ejemplo interesante del uso de colas es para el algoritmo de ordenación llamado Radix Sort que veremos más adelante.

También veremos cómo se puede implementar el TAD cola.

TAD cola de prioridades. Es similar a la cola, pero en vez de utilizarse el orden de llegada a la cola como criterio para establecer el orden de atención, se asume que los elementos son prioridades. Se atiende a los elementos según su prioridad, primero el mayor elemento y luego los menores. Por ello, la operación primero devuelve el de mayor prioridad y la operación decolar lo remueve.

TAD pcola[elem,$\leq$]
constructores
  vacía : pcola
  encolar : pcola $\times$ elem $\rightarrow$ pcola
operaciones
  primero : pcola $\rightarrow$ elem {se aplica sólo a una pcola no vacía}
  decolar : pcola $\rightarrow$ pcola {se aplica sólo a una pcola no vacía}
  es_vacía : pcola $\rightarrow$ booleanos
ecuaciones
  primero(encolar(vacía,e)) = e
  e $\geq$ e' $\Rightarrow$ primero(encolar(encolar(q,e'),e)) = primero(encolar(q,e))
  e $<$ e' $\Rightarrow$ primero(encolar(encolar(q,e'),e)) = primero(encolar(q,e'))
  decolar(encolar(vacía,e)) = vacía
  e $\geq$ e' $\Rightarrow$ decolar(encolar(encolar(q,e'),e)) = encolar(decolar(encolar(q,e)),e')
  e $<$ e' $\Rightarrow$ decolar(encolar(encolar(q,e'),e)) = encolar(decolar(encolar(q,e')),e)
  es_vacía(vacía) = verdadero
  es_vacía(encolar(q,e)) = falso

Además del parámetro elem, se asume que se dispone de una relación de orden total entre los elementos de elem. Por primera vez se utilizan ecuaciones condicionales.

Desde el punto de vista conceptual, debe observarse que el orden en que se encolan los elementos es irrelevante: es una cola en lo que lo único que interesa son las prioridades, no el orden de llegada. Por lo tanto, si los elementos ingresan en un orden u otro, la cola es la misma. Esto puede expresarse con la siguiente ecuación entre constructores: encolar(encolar(q,e),e') = encolar(encolar(q,e'),e).

Al agregarse esta ecuación, la 3era y 6ta ecuación de la especificación de arriba resultan redundantes ya que se deducen de las demás ecuaciones:

TAD pcola[elem,$\leq$]
... *acá se repiten los constructores y operaciones de la especificación anterior* ...
ecuaciones
  encolar(encolar(q,e),e') = encolar(encolar(q,e'),e)
  primero(encolar(vacía,e)) = e
  e $\geq$ e' $\Rightarrow$ primero(encolar(encolar(q,e'),e)) = primero(encolar(q,e))
  decolar(encolar(vacía,e)) = vacía
  e $\geq$ e' $\Rightarrow$ decolar(encolar(encolar(q,e'),e)) = encolar(decolar(encolar(q,e)),e')
  es_vacía(vacía) = verdadero
  es_vacía(encolar(q,e)) = falso

Se verá más adelante que las colas de prioridades se implementan eficientemente utilizando una estructura de datos llamada heap.