

Tipos abstractos de datos

- **Tipos concretos:**
 - Son nativos del lenguaje.
 - Tipos básicos: enteros, booleanos, char, etc.
 - Tipos más complejos: arreglos, punteros, tuplas.
 - En general conocemos cómo están implementados en el lenguaje.
- **Tipos abstractos:**
 - Se definen **especificando** constructores y operaciones.
 - Podemos tener varias **implementaciones** para un mismo TAD.
 - En general surgen de analizar un problema a resolver.
 - El problema evidencia qué necesitamos representar y qué operaciones tener.

Especificación

Para especificar un TAD debemos:

- Indicar su nombre
- Especificar constructores: procedimientos o funciones mediante los cuales puedo crear elementos del tipo que estoy especificando.
- Especificar operaciones: todos los procedimientos o funciones que permitirán manipular los elementos del tipo de datos que estoy especificando.
- Indicamos los tipos de cada constructor y operación (el encabezado de los procedimientos o funciones), y mediante lenguaje natural explicamos qué hacen.
- Algunas operaciones pueden tener restricciones que las indicamos mediante **precondiciones**.
- Debemos especificar también una operación de destrucción que libera la memoria utilizada por los elementos del tipo, en caso que sea necesario.

Implementación

A partir de una especificación de un TAD, para implementarlo debemos:

- Definir un nuevo tipo con el nombre del TAD especificado. Para ello utilizamos tipos concretos y otros tipos definidos previamente.
- Implementar cada constructor respetando los tipos tal como fueron especificados.
- Implementar cada operación respetando los tipos tal como fueron especificados.
- Implementar operación de destrucción liberando memoria si es que se ha reservado al construir los elementos.
- Pueden surgir nuevas restricciones que dependen de cómo implementamos el tipo.
- Puedo necesitar operaciones auxiliares que no están especificadas en el tipo.

Listas

- Las listas permiten resolver una gran cantidad de problemas.
- Son colecciones de elementos de un mismo tipo, de tamaño variable.
- Toda lista o bien es vacía o bien tiene al menos un elemento al comienzo.
- Operaciones:
 - decidir si una lista es vacía
 - tomar el primer elemento
 - tirar el primer elemento
 - agregar un elemento al final
 - obtener la cantidad de elementos
 - concatenar dos listas
 - obtener el elemento en una posición específica
 - tomar una cantidad arbitraria de elementos
 - tirar una cantidad arbitraria de elementos
 - copiar una lista en una nueva

Especificación de listas

spec List of T where

constructors

```
fun empty() ret l : List of T
{- crea una lista vacía. -}
```

```
proc addl (in e: T, in/out l: List of T)
{- agrega el elemento e al comienzo de la lista l. -}
```

destroy

```
proc destroy (in/out l: List of T)
{- Libera memoria en caso que sea necesario. -}
```

operations

```
fun is_empty(l: List of T) ret b : bool
{- Devuelve True si l es vacía. -}
```

```
fun head(l: List of T) ret e: T
{- Devuelve el primer elemento de la lista l -}
{- PRE: not is_empty(l) -}
```

```
proc tail(in/out l: List of T)
{- Elimina el primer elemento de la lista l -}
{- PRE: -is_empty(l) -}
```

```
proc addr(in/out l: List of T, in e: T)
{- agrega el elemento e al final de la lista l. -}
```

```
fun length(l: List of T) ret n: nat
{- Devuelve la cantidad de elementos de la lista l -}
```

```
proc concat(in/out l: List of T, in l0: List of T)
{- Agrega al final de l todos los elementos de l0 en el mismo orden.-}
```

```

fun index(l: List of T, n: nat) ret e: T
{- Devuelve el n-ésimo elemento de la lista l -}
{- PRE: length(l) > n -}

proc take(in/out l: List of T, in n: nat)
{- Deja en l sólo los primeros n elementos, eliminando el resto -}

proc drop(in/out l: List of T, in n: nat)
{- Elimina los primeros n elementos de l -}

fun copy_list(l1: List of T) ret l2: List of T
{- Copia todos los elementos de l1 en la nueva lista l2 -}
end spec

```

- Para **usar** desde algún programa el tipo de las listas, alcanza con su especificación.
- Mediante sus **constructores** empty y addl pueden crearse listas vacías o agregar a una lista un elemento nuevo, respectivamente.
- Las **operaciones** permiten manipular las listas de acuerdo a la funcionalidad que el TAD provee.
- **No** es necesario conocer la **implementación** para poder **usar** el TAD.

Ejemplo de uso del TAD lista

```

fun promedio (l: List of float) ret r: float
  var largo: nat
  var elem: float
  var laux: List of float

  laux:= copy(l)
  r:= 0.0
  largo:= length(l)

  while ¬is_empty(laux) do
    elem:= head(laux)
    r:= r + elem
    tail(laux)
  od
  destroy(laux)
  r:= r div largo
end proc

```

Implementación de listas mediante punteros

- Implementaremos el TAD lista utilizando punteros, implementación conocida como **lista enlazada**.
- Cada elemento de la lista estará alojado en un nodo conteniendo además un puntero hacia el siguiente.
- Una lista será un puntero a un nodo.
- La lista vacía se implementa con el puntero **null**.

- Esta implementación permite tener la lista de elementos almacenada en lugares de la memoria no necesariamente contiguos.
- No existe límite teórico para almacenar elementos. En la práctica dicho límite será la cantidad de memoria.

```

implement List of T where
  type Node of T = tuple
    elem: T
    next: pointer to (Node of T)
  end tuple

  type List of T = pointer to (Node of T)

  fun empty() ret l: List of T
    l := null
  end fun

```

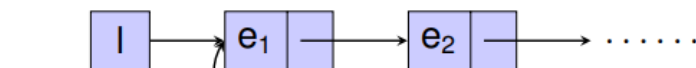
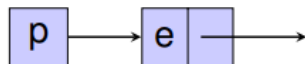
Implementación de listas mediante punteros: addl



```

alloc(p)
p -> elem := e

```



```

p->next := l

```



```

l := p

```



```

proc addl (in e: T, in/out l: List of T)
  var p: pointer to (Node of T)
  alloc(p)
  p -> elem := e
  p -> next := l
  l := p
end proc

```

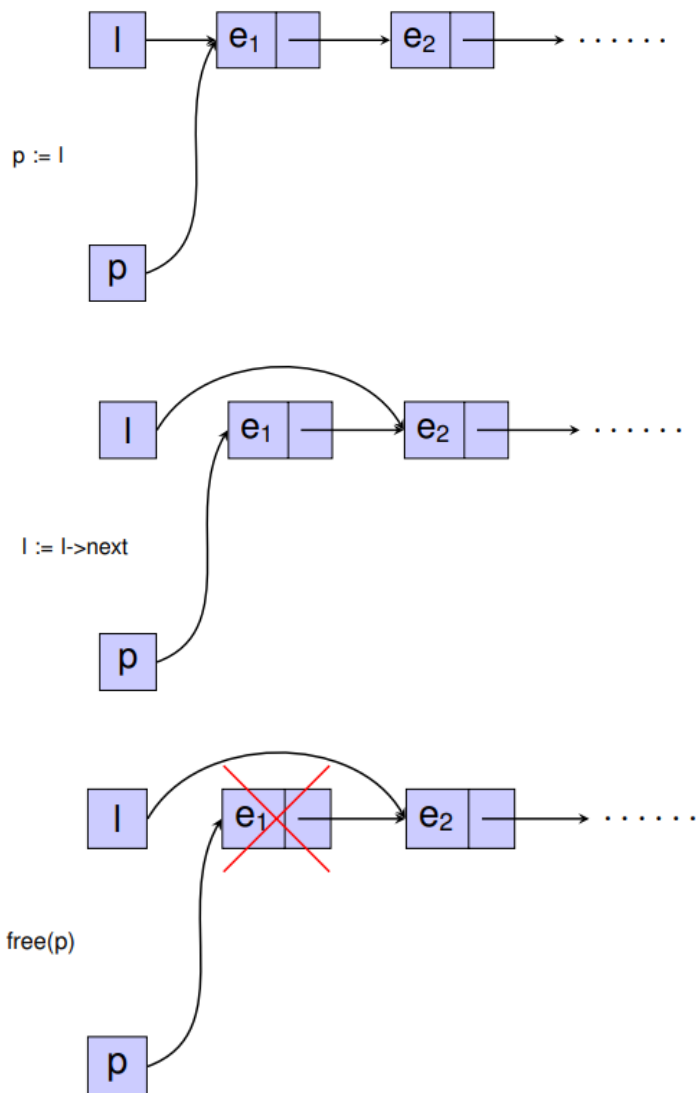
```

fun is_empty(l: List of T) ret b: bool
    b := l = null
end fun

{- PRE:  $\neg$ is_empty(l) -}
fun head(l: List of T) ret e: T
    e := l -> elem
end fun

```

Implementación de listas mediante punteros: tail

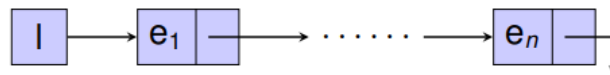


```

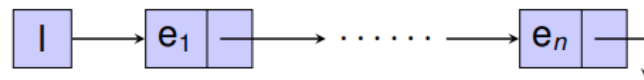
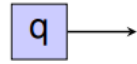
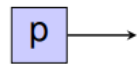
{- PRE:  $\neg$ is_empty(l) -}
proc tail(in/out l: List of T)
    var p: pointer to (Node of T)
    p := l
    l := l -> next
    free(p)
end proc

```

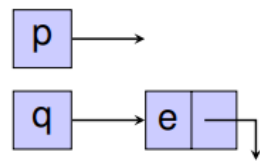
Implementación de listas mediante punteros: addr



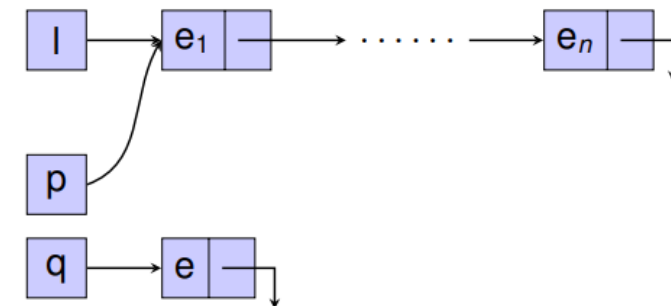
var p, q : pointer to Node of T



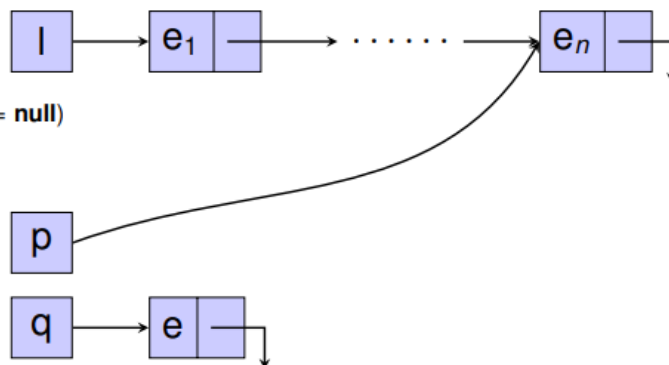
alloc(q)
q->elem := e
q->next := null



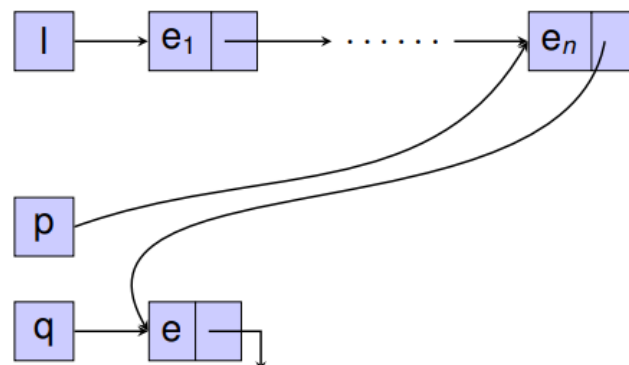
p := l



do (p->next != null)
p := p->next
od



p->next := q



```

proc addr (in/out l: List of T, in e: T)
  var p,q: pointer to (Node of T)
  alloc(q) q -> elem:= e
  q -> next:= null
  if is_empty(l) then
    p:= l
    while p -> next != null do
      p:= p -> next
    od
    p -> next:= q
  else
    l:= q
  fi
end proc

```

```

fun length (l: List of T) ret n: nat
  var p: pointer to (Node of T)
  n:= 0
  p:= l
  while p != null do
    n:= n + 1
    p:= p -> next
  od
end fun

```

TAD contador

Problema:

- 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.

Solución conocida:

- Recorrer el arreglo de izquierda a derecha,
- utilizando un entero **inicializado en 0**,
- **incrementarlo** cada vez que se encuentra un paréntesis que abre,
- **decrementarlo** (comprobando previamente que no sea nulo en cuyo caso no están balanceados) cada vez que se encuentra un paréntesis que cierra,
- Al finalizar, **comprobar** que dicho entero sea cero.
- ¿Es necesario que sea un entero?

Contador:

- No hace falta un entero (susceptible de numerosas operaciones aritméticas),
- sólo se necesita **algo** con lo que se pueda
 - inicializar
 - incrementar
 - comprobar si su valor es el inicial
 - decrementar si no lo es
- Llamaremos a ese **algo, contador**
- Necesitamos un contador.

- El contador se define por lo que sabemos de él: sus cuatro operaciones
 - inicializar
 - incrementar
 - comprobar si su valor es el inicial
 - decrementar si no lo es
- Notamos que las operaciones **inicializar** e **incrementar** son capaces de generar todos los valores posibles del contador, por lo que serán nuestros **constructores**.
- **comprobar** en cambio solamente examina el contador,
- **decrementar** no genera más valores que los obtenibles por **inicializar** e **incrementar**

Especificación

spec Counter where

constructors

```
fun init() ret c: Counter
{- crea un contador inicial. -}
```

```
proc incr(in/out c: Counter)
{- incrementa el contador c. -}
```

destroy

```
proc destroy(in/out c: Counter)
{- Libera memoria en caso que sea necesario. -}
```

operations

```
fun is_init(c: Counter) ret b: Bool
{- Devuelve True si el contador es inicial -}
```

```
proc decr(in/out c: Counter)
{- Decrementa el contador c. -}
{- PRE: ¬is_init(c) -}
```

end spec

Resolviendo el problema:

- Queremos implementar un algoritmo que resuelve el problema de los paréntesis balanceados utilizando el TAD contador.
- La especificación nos da toda la información que necesitamos tener: constructores y operaciones con sus tipos.
- La idea es iniciar un contador y recorrer el arreglo de caracteres de izquierda a derecha.
- Si encontramos un paréntesis que abre, incrementamos el contador.
- Si encontramos un paréntesis que cierra lo decrementamos.
- Si el contador es inicial y encuentro paréntesis que cierra devuelvo False. Si termino de recorrer el arreglo y el contador no es inicial también doy False.


```

fun matching_parenthesis(a: array[1..n] of char) ret b: bool
    var i: nat
    var c: Counter
    b:= true
    init(c)
    i:= 1
    while i ≤ n ∧ b do
        if a[i] = '(' then
            inc(c)
        else if a[i] = ')' ∧ is_init(c) then
            b:= false
        else if a[i] = ')' ∧ ¬is_init(c) then
            dec(c)
        else
            skip
        fi
        i:= i+1
    od
    b:= b ∧ is_init(c)
    destroy(c)
end fun

```

Implementación

```

implement Counter where
    type Counter = nat

    proc init(out c: Counter)
        c:= 0
    end proc

    proc inc(in/out c: Counter)
        c:= c+1
    end proc

    fun is_init(c: Counter) ret b: bool
        b:= (c = 0)
    end fun
    {- PRE: ¬is_init(c) -}

    proc dec(in/out c: Counter)
        c:= c-1
    end proc

    proc destroy (in/out c: Counter)
        skip
    end proc

```