0. INTRODUCCIÓN

0.1. GENERALIDADES

Tipos elementales de datos.

var j, k : integer; int j, k;

c : char; char c;

x : real; float x;

p : ^integer; int *p;

Punteros y nodos.

type enlace = ^nodo; typedef struct node

nodo = record { int dato;

dato : integer; struct node *link;

link : enlace; } nodo;

end; typedef nodo *enlace;

var q : enlace; enlace q;

0.2. LISTAS Y RECURSIVIDAD

Escribir un programa que genere una lista lineal de enlace simple de valores enteros. Utilizar un subprograma para almacenar cada valor recibido en un nuevo nodo y agregar ese nodo en el "extremo izquierdo" de la lista.

En Pascal

program Uno(Input, Output);

type Enlace = ^Nodo;

Nodo = record

dato : integer;

link : enlace;

end;

var q : Enlace;

k : integer;

```
procedure Agregar(var l : Enlace; e : integer);

var p : Enlace;

begin

New(p);

p^.dato := e;

p^.link := l;

l := p:

end;

begin

q := nil;

Read(k);

while k <> 0 do

begin

Agregar(q,k);

Read(k);

end;

end.
```

En C

```
#include <stdio.h>

#include <conio.h>

typedef struct node

{ int dato;

struct node *link;

} nodo;

typedef nodo *enlace;

void agregar(enlace *l, int e)
```

```
{ enlace p;

p = malloc(sizeof(nodo));

p−>dato = e;

p−>link = *l;

*l = p;

}

void main()

{ int k;

enlace q;

q = NULL;

scanf("%d", &k);

while (k != 0)

{ agregar(&q, k);

scanf("%d", &k);

}

}
```

Escribir un programa que genere una lista lineal de enlace simple de valores enteros. Utilizar un subprograma para almacenar cada valor recibido en un nuevo nodo y agregar ese nodo en el "extremo derecho" de la lista.

En Pascal

```
program Dos(Input, Output);

type Enlace = ^Nodo;

Nodo = record

dato : integer;

link : Enlace;

end;

var q : Enlace;

k : integer;
```

```
procedure Agregar(var l : Enlace; e : integer);
begin
if l = nil then
begin
New(l);
l^.dato := e;
l^.link := nil;
end
else Agregar(l^.link,e);
end;
begin
q := nil;
Read(k);
while k <> 0 do
begin
Agregar(q,k);
Read(k);
end;
end.
```

En C

```
#include <stdio.h>
#include <conio.h>
typedef struct node
{ int dato;
struct node *link;
} nodo;
```

```
typedef nodo *enlace;

void agregar(enlace *l, int e)

{ if (*l == NULL)

{ *l = malloc(sizeof(nodo));

(*l)−>dato = e;

(*l)−>link = NULL;

}

else agregar(&((*l)−>link),e);

}

void main()

{ int k;

enlace q = NULL;

scanf("%d", &k);

while (k != 0)

{ agregar(&q, k);

scanf("%d", &k);

}

}
```

Implementar una función que invierta el acceso y el sentido de los enlaces de una lista lineal de enlace simple l.

```
function Invierte(p, q : Enlace) : Enlace;

begin

if p <> nil then

begin

Invierte := Invierte(p^.link,p);

p^.link := q;

q^.link := nil;
```

```
end

else Invierte := q;

end; {Invocación : Invierte(l,l)}

lista invierte(lista p, lista q)

{ lista l;

if(p != NULL)

{ l = invierte(p−>link,p);

p−>link = q;

q−>link = NULL;

return l;

}

else return q;

} {Invocación : invierte(l,l)}
```

Implementar un procedimiento que invierta el acceso y el sentido de los enlaces de una lista lineal de enlace simple l.

```
procedure Invierte(p, q : Enlace; var l : Enlace);

begin

if p <> nil then

begin

Invierte(p^.link,p,l);

p^.link := q;

q^.link := nil;

end

else l:= q;

end; {Invocación : Invierte(l,l,l)}

void invierte(enlace p, enlace q, enlace *l)

{ if(p != NULL)
```

```
{ invierte(p−>link,p,l);

p−>link = q;

q−>link = NULL;

}

else *l = q;

} {Invocación : invierte(l,l,&l)}
```

0.3. MULTILISTAS

Una *multilista* es una lista generalizada cuyos elementos pueden ser *átomos* o *multilistas*. Por ejemplo, si M = (1 2 (3 4 (5) 6) 7 8) es una multilista, entonces M consta de cinco elementos: los *átomos* 1, 2, 7, 8 y la *multilista* (3 4 (5) 6) la cual, a su vez, consta de cuatro elementos.

Implementación de una *multilista*:

En Pascal

```
type Mulista = ^Nodo;

Nodo = record

link : Mulista;

case atomo : boolean of

true : (dato : integer);

false : (next : Mulista);

end;
```

En C

```
typedef struct node

{ int atomo;

union

{ int dato;

struct node *next;

} var;

struct node *link;
```

```
} nodo;

typedef nodo *mulista;
```

Implementar, en Pascal, un operador para imprimir el contenido de una multilista:

```
procedure Listar(m : Mulista);

begin

if m <> nil then

if m^.atomo then

begin

Write(m^.dato)

Listar (m^.link);

end

else

begin

Write('(');

Listar(m^.next);

Write(')');

Listar (m^.link);

end;

end;
```

Implementar, en lenguaje C, operadores para:

Generar una *multilista* m.

Imprimir el contenido de m incorporando los paréntesis necesarios.

Usar los dos operadores anteriores.

```
void generar(mulista *m)

{ mulista p;

int e;
```

```
printf("Valor '(' = 0, ')' = −1: FIN: −1 ");
scanf("%d", &e) ;
if (e == −1)
*m = NULL;
else
if (e == 0)
{ p = malloc(sizeof(nodo));
*m = p;
p−>atomo = 0;
p−>link = NULL;
generar(&(p−>var.next));
generar(&(p−>link));
}
else
{ p = malloc(sizeof(nodo));
*m = p;
p−>atomo = 1;
p−>var.dato = e;
generar(&(p−>link));
};
};
void listar(mulista m)
{ if (m != NULL)
if (m−>atomo)
{ printf("%d", m−>var.dato);
listar(m−>link);
```

```
}

else

{ printf("(");

listar(m−>var.next);

printf(")");

listar(m−>link);

}

};

void main()

{ mulista l;

int k;

generar(&l);

listar(l);

printf("%d", k);

}
```

0.4. ÁRBOLES DE EXPRESIONES

Un árbol binario de expresión (ABE) es un árbol binario utilizado para representar una expresión aritmética de la siguiente manera:

Cada hoja contiene un operando.

Cada nodo interior contiene un operador aritmético.

Un ABE tiene información implícita sobre prioridad de operadores y asociatividad.

Sea T un ABE compuesto de un nodo N y subárboles izquierdo Ti y derecho Td. La evaluación de T se efectúa de acuerdo al siguiente algoritmo:

Si Ti y Td son árboles vacíos, entonces el valor de T es el valor contenido en N.

Si Ti y Td son árboles no vacíos, entonces el valor de T es el valor de Ti operado con el valor de Td según el operador contenido en N.

Implementación de un ABE:

En Pascal

```
type Arbex = ^Nodo;

Nodo = record

izq : Arbex;

der : Arbex;

case hoja : boolean of

true : (valor : real);

false : (signo : char);

end;
```

En C

```
typedef struct node

{ int hoja;

union

{ float valor;

char signo;

} var;

struct node *izq;

struct node *der;

} nodo;

typedef nodo *arbex;
```

Con respecto a un árbol binario de expresión t, implementar en lenguajes Pascal y C operadores para:

Evaluar la expresión que t representa.

Imprimir el contenido de t incorporando los paréntesis necesarios.

```
function Evalua(t : Arbex) : real;

begin

if t <> nil then

if t^.hoja then
```

```
Evalua := t^.valor

else

case t^.signo of

'+' : Evalua := Evalua(t^.izq) + Evalua(t^.der);

' '" : Evalua := Evalua(t^.izq) " Evalua(t^.der);

'*' : Evalua := Evalua(t^.izq) * Evalua(t^.der);

'/' : Evalua := Evalua(t^.izq) / Evalua(t^.der);

end;

end;

float evalua(arbex t)

{ if (t != NULL)

if (t−>hoja)

return t−>var.valor;

else

switch (t−>var.signo)

{ case '+' : return evalua(t−>izq) + evalua(t−>der);

case "" : return evalua(t−>izq) " evalua(t−>der);

case "" : return evalua(t−>izq) " evalua(t−>der);

case '/' : return evalua(t−>izq) / evalua(t−>der);

}

}

procedure Imprime(t : Arbex);

begin

if t <> nil then

begin

if t^.hoja then
```

```
Write(t^.valor)
else
begin
Write('(');
Imprime(t^.izq);
Write(t^.signo);
Imprime(t^.der);
Write(')');
end;
end;
end;
void imprime(arbex t)
{ if (t != NULL)
if (t−>hoja)
printf(t−>var.valor);
else
{ printf ("(");
imprime(t−>izq);
printf ("%c", t−>var.signo);
imprime(t−>der);
printf (")");
}
}
```

1. TIPOS DE DATOS

1.1. DEFINICIONES FORMALES

Tipo de dato

Un tipo de dato es un conjunto de objetos con una colección de operaciones. Tal conjunto se conoce como dominio.

Tipo elemental de dato

También conocido como escalar, es aquel cuyo dominio consta sólo de valores constantes (enteros, reales, lógicos y caracteres).

Tipo estructurado de dato

También conocido como agregado, es aquel en el cual cada uno de los valores de su dominio es una composición de objetos de tipo escalar y/o agregado.

Producto cartesiano

El producto cartesiano de n conjuntos C1, C2, ... Cn, denotado en la forma C1 x C2 x ... x Cn, es un conjunto cuyos elementos son n−tuplas (c1, c2, ... cn) donde ci ð Ci. Por ejemplo, los polígonos regulares se pueden caracterizar por un número entero que representa el número de lados y un número real que representa la longitud de un lado. Todo polígono regular así expresado sería un elemento del producto cartesiano **Z**x**R**.

A los productos cartesianos se les conoce como registros o estructuras.

Aplicación finita

Una aplicación finita es una función de un conjunto de valores pertenecientes a un dominio **D** sobre un conjunto de valores pertenecientes a una imagen **I**.

Las aplicaciones finitas se conocen como arreglos. En Pascal,

var **a**: array [1..5] of real;

define una aplicación del subrango de enteros 1...5 sobre los números reales, pudiéndose seleccionar mediante un índice un objeto del conjunto imagen, por ejemplo **a**[k], 1 " k " 5.

La estrategia para ligar el dominio de la función a un subconjunto específico de valores depende del lenguaje presentándose tres alternativas: (ligar imagen a dominio y establecer cardinalidad del dominio).

" Estática

El subconjunto se determina en tiempo de compilación.

" Semidinámica

El subconjunto se determina en tiempo de creación del objeto. Por ejemplo,

[m:n] int **a**;

declara un arreglo semidinámico cuyo espacio en memoria se asigna según los valores actuales de m y n y permanece mientras a exista en el ambiente en el que fue definido.

" Dinámica

El subconjunto puede cambiar en cualquier instante de la existencia del objeto. Por ejemplo

flex [1:0] int **b**;

declara un arreglo vacío de modo que, la asignación **b**:= (2, 3, 8) cambia sus límites a [1:3].

Unión discriminada

La unión discriminada constituye una extensión del producto cartesiano, cuyo propósito es permitir, en cualquier instante de la ejecución del código al que pertenece, la elección de una entre diferentes estructuras alternativas, cada una de las cuales se denomina **variante**, dependiendo del valor que presenta un selector conocido como **discriminante**.

Conjunto potencia

Corresponde a un mecanismo de estructuración que permite definir variables cuyo valor puede ser cualquier subconjunto de un conjunto de elementos de un determinado tipo T. El tipo de tales variables es el conjunto de todos los subconjuntos de elementos de tipo T, conocido como tipo base.

Si T = {a, b, c} entonces el conjunto potencia de T es

P (T) = { , {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}} y

#(P (T)) = 2#(T) = 23 = 8

1.2. MODELOS DE CONSTRUCCIÓN

Caracterización

Un **modelo de construcción** es una técnica destinada a una futura creación de objetos en memoria.

Los traductores de lenguajes operan con tales modelos con el propósito de satisfacer abstracciones de alto nivel.

Un **modelo de construcción** está formado por un par consistente en un **descriptor** y un **modelo de representación**.

El **descriptor** es un conjunto de atributos del objeto.

El **modelo de representación** corresponde a la forma física que adoptará el objeto en memoria.

Al interior del **descriptor**, el **constructor** expresa el **modelo de representación** que adoptará el objeto.

El **constructor** es un **código ejecutable** destinado a crear la representación del objeto en memoria.

A partir del **modelo de construcción**, el compilador genera una función de transformación de referencias lógicas en físicas, cuya forma concreta es un **código ejecutable** y cuya forma abstracta es una **fórmula de acceso**.

Modelos de tipo y clase:

modelo de tipo = definición + modelo de construcción

modelo de clase = definición + implementación + modelo de construcción

modelo de construcción = descriptor + modelo de representación

Tipos elementales

En casi todos los lenguajes, los tipos elementales constituyen una abstracción de una implementación hardware. Sin embargo, ciertos lenguajes definen como elementales algunos tipos para los cuales no existe representación hardware, debiendo simularse mediante software.

Producto cartesiano

La representación interna del producto cartesiano (modalidad Pascal) consta de un descriptor y de la disposición secuencial de sus componentes (campos) en un bloque.

Por ejemplo

var R : record

a : integer;

b : real;

end;

se representa, gráficamente, de la siguiente forma:

Descriptor Objeto

El acceso al *i*−ésimo selector, durante ejecución, se expresa mediante la fórmula:

Dir(R.*Si*) = +