Modula-2

Daniel Delgado, *Estudiante, ITCR,* Wilbert Gonzales, *Estudiante, ITCR,*
Anthony Leandro, *Estudiante, ITCR,* and Bryan Mena, *Estudiante, ITCR*

1. Datos Historicos

Modula-2 fue creado en 1978 por Niklaus Wirth. Inicio como una revisión de *Pascal* para funcionar en el sistema operativo de una computadora llamada Lilith. Los tres conceptos fundamentales de este lenguaje de programación son: Utilización del modulo como una unidad de compilación, con el fin de tener una compilación separada. Existen tipos y procedimientos que permiten el acceso a datos de la maquina. *Modula-2* es utilizado en su mayoria por universidades ya que tienen una gran adaptacion a la enseñanza de lenguajes estructurados.

Figura 1. Niklaus Wirth.

2. Tipos de Datos

En *Modula-2* se utiliza la palabra *VAR* cuando se va a realizar una declaración de una variable.

Nombre	Tipo de Dato	Ejemplo
Integer	Número Entero	-32768 a 32767
Cardinal	Número Entero	0 a 65.535
Real	Número Real	8.43E-37 a 3.37E+38
Char	Un solo carácter	'S', 'A'
Boolean	Valor Booleano	TRUE, FALSE

2.1. Scalars

También se conocen como *enumerated*, es una lista de valores que una variable de este tipo puede asumir. *Dias* es un tipo que puede asumir cualquiera de los siete valores. Dentro de la declaración *VAR* a *Dia* se le asigna el tipo *Dias* entonces *Dia* puede asumir cualquiera de los siete diferentes valores, lo que hace que el programa sea fácil de seguir y entender. Es importante de saber que el sistema *Modula-2* no asigna el valor *l* a la variable *Dia* pero usa una representación entera para cada nombre, la importancia de esta característica se debe a que no se puede imprimir *l, k, etc.*, sino que se pueden usar para indexar instrucciones de control.

Cuadro 1. Código de declaración para un Scalar

```
MODULE Entypes;

FROM Terminal2
IMPORT WriteString, WriteLn, WriteReal;

TYPE
Dias      = (l,k,m,j,v,s,d);
TiempoDia = (mañana,tarde,noche);

VAR
Dia              : Dias;
Tiempo   : TiempoDia;

BEGIN    (* Estructura del programa *)
.
.
.
END Entypes.
```

Cuadro 2. Código de declaración para un Scalar

```
TYPE
<Nombre Tipo> =
  (<identificador>,...,<identificador>);
```

2.2. Arrays

Una lista esta compuesta por elementos del mismo tipo. En la definición de una lista se usa la palabra reservada *ARRAY*, seguidamente por corchetes con un rango y de un tipo de dato. En el código de ejemplo se define una lista de 12 diferentes números de tipo *CARDINAL*. Para acceder a los números de la lista se usa:

Autos [1], Autos [2], ... Autos [12]
Ademas, de ver los valores se puede asignar valores o se pueden utilizar para cálculos o en algún lugar del programa donde sea legal el uso de una variable de tipo *CARDINAL*. En un lugar donde no se puede usar es en un indice de un bucle *FOR* ya que es obligatorio un tipo de variable simple.

Cuadro 3. Código de declaración para un array

```
MODULE Arrays;

FROM Terminal2
IMPORT WriteString, WriteCard, WriteLn;

VAR
Index : CARDINAL;
Autos : ARRAY [1..12] OF CARDINAL;
BEGIN   (* Estructura del programa *)
.
.
.
END Arrays.
```

2.3. Matrix

En la sección *VAR* se define la variable de tipo *ARRAY* que contiene 8 elementos en el que cada elemento es otro *ARRAY* de 8 elementos y se obtiene una matriz de 8 por 8 en la cual se pueden almacenar una variable de tipo *CARDINAL*. La variable *Valor* se define de la misma manera, pero el método de definición es un poco diferente. Los 2 métodos resultan del mismo tipo y número de variables.

Cuadro 4. Código de declaración para una matriz

```
MODULE Arrays2;

FROM Terminal2
IMPORT WriteString, WriteInt, WriteLn;

VAR
Indice, Cont     : CARDINAL;
Tablero: ARRAY[1..8] OF ARRAY[1..8]
                     OF CARDINAL;
Valor   : ARRAY[1..8],[1..8] OF CARDINAL;

BEGIN (* Estructura del programa *)
.
.
.
END Arrays2.
```

2.4. Punteros

Como en C o C++ un puntero es una dirección a memoria, de manera similar es necesario indicar a que tipo de dato se apuntará, luego se debe asignar un espacio en memoria que será donde el puntero viva, esto se hace con el comando:

Cuadro 5. Código de una lista con punteros

```
MODULE Pointers;

FROM Terminal2
IMPORT WriteString, WriteInt, WriteLn;
FROM Storage
IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM
IMPORT TSIZE;

TYPE
Nombre = ARRAY[0..20] OF CHAR;

VAR
MiNbr : POINTER TO Nombre;
(* MiNbr apunta a un String *)
MiEdad  : POINTER TO INTEGER;
(* MiEdad apunta a un entero *)

BEGIN (* Estructura del programa *)
.
.
.
END Pointers.
```

Para asignarle un valor al espacio de memoria al cual se esta apuntando se necesita utilizar de nuevo el carácter ^ y asignar el valor, pero para esto primero se debe de usar el comando ALLOCATE(NombrePuntero, TSIZE(TipoDatoApuntado)) que se utiliza para reservar memoria y al final del programa se debe de utilizar el comando DEALLOCATE(NombrePuntero, TSIZE(TipoDatoApuntado)) que es para liberar la memoria. Por ejemplo:

Cuadro 6. Código de asignacion a un puntero.

```
MODULE Pointers;

FROM Terminal2
IMPORT WriteString, WriteInt, WriteLn;
FROM Storage
IMPORT ALLOCATE, DEALLOCATE;
FROM SYSTEM
IMPORT TSIZE;

TYPE
Nombre = ARRAY[0..20] OF CHAR;

VAR
MiNbr : POINTER TO Nombre;
(* MiNbr apunta a un String *)
MiEdad  : POINTER TO INTEGER;
(* MiEdad apunta a un entero *)

BEGIN
ALLOCATE(MiEdad, TSIZE(INTEGER));
ALLOCATE(MiNbr, TSIZE(Nombre));

MiEdad^  := 20;
MiNbr^ := "Willy";
```

```
WriteString("Mi nombre es ");
WriteString(MiNbr^);
WriteString(" y tengo ");
WriteInt(MiEdad^,2);
WriteLn;

DEALLOCATE(MiEdad,TSIZE(INTEGER));
DEALLOCATE(MiNbr,TSIZE(Nombre));
END Pointers.
```

2.5. Subrangos

Primero se define un Tipo *Dias* que es un *Scalar*, luego se define *Trabajar* donde se toman los valores del *Scalar* del *L al V* . Para usar estos Subrangos se deben de definir el la sección *VAR*.

Cuadro 7. Ejemplo de subrangos

```
MODULE Subrange;

TYPE
Dias = (L,K,M,J,V,S,D);
Trabajar = [L..V];
Descansar = [S..D];

VAR
Dia : Dias;
(* Puede ser cualquier dia *)
DiaTrabajo : Trabajar;
(* Son los dias de L a V  *)
FinSemana : Descansar;
(* Solo los 2 dias de Descansar *)

BEGIN (* Estructura del programa *)
.
.
.
END Subrange.
```

2.6. Puntos Importantes[1]

A continuación se presentan algunos de los rangos para *Integers*.

Tipo	Rango	Bytes
ShortInt	-128..127	1
ShortCard	0..255	1
Integer8	-128..127	1
Cardinal8	0..255	1
Integer16	-32768..32767	2
Cardinal16	0..65535	2
Integer32	-2147483648..2147483647	4
Cardinal32	0..4,294,967,295	4
Integer64	$-2^{63}..(2^{63})-1$	8
Cardinal64	$0..(2^{63})-1$	8
LongInt	-2,147,483,648..2,147,483,647	4
LongCard	$0..(2^{64})-1$	8

3. Estructuras de control y Expresiones

En *Modula-2* se utilizan bloques de código, inician con un *begin* y terminan con un *end*. Como dato curioso en Modula-2 *end* no necesariamente termina en *;* al contrario cuando se utiliza para terminar un archivo de *Modula-2* se utiliza un . y el nombre del modulo.
Ejemplo:

Cuadro 8. Código Hola Mundo en Modula-2

```
MODULE PrimerEjemplo;

FROM InOut
IMPORT WriteString, WriteLn;

BEGIN
        WriteString('Hola Mundo!');
        WriteLn;
END PrimerEjemplo.
```

3.1. Estructura típica de un programa

Cuadro 9. Estructura típica de un programa en Modula-2

```
MODULE <Nombre del programa> ;

FROM <Libreria> IMPORT <Funciones>;

CONST
        <Declarar Constantes>

TYPE
```

1. Datos Obtenidos de *La programacin imperativa desde Modula-2*, Proyectos y Producciones Editoriales Cyan.

```
        <Declarar Tipos>

VAR
        <Declarar Variables>

BEGIN
        <Expresiones Ejecutables>

END <Nombre del programa>.
```

3.2. Constantes

Se definen en el bloque de constantes al inicio del programa, con la palabra reservada *CONST* y el valor que almacenan no podrá ser cambiado a lo largo del programa.

Cuadro 10. Declaración de Constantes

```
MODULE Constant;

CONST
Max    = 12;
Indice = 49;

TYPE
String = ARRAY[1..Max] OF CHAR;

VAR
Vacas      : String;
Caballos   : String;

BEGIN      (* Estructura del programa *)
.
.
.
END Constant.
```

3.3. Variables

Cuadro 11. Declaración de Variables

```
MODULE IntVar;

FROM Terminal2
IMPORT WriteLn, WriteString, WriteInt;

VAR
(* Declaracion de variables en este bloque *)
Count : INTEGER;
x,y   : INTEGER;

BEGIN
.
.
.
END IntVar.
```

Cabe mencionar que los identificadores separados por comas serán declarados del mismo tipo.

3.4. Type

En la parte superior del código tenemos un grupo de declaraciones *TYPE*. La primera definición *DefLista* se utiliza de la misma manera que se utilizaría un *Integer* o cualquier otra definición simple. La variable llamada *Aux* se define como una variable de tipo *DefLista* y como *DefLista* es un *ARRAY* de 14 elementos, entonces *Aux* es un array de 14 elementos de *INTEGER*. La variable *Avion* es una lista de 12 elementos del tipo *Comida*.

Cuadro 12. Código de declaración para un tipo

```
MODULE Types;

TYPE
DefLista = ARRAY[12..25] OF INTEGER;
DefChar = ARRAY[0..27] OF CHAR;
ArrayReal = ARRAY[-17..42] OF REAL;
Comida = ARRAY[1..6] OF BOOLEAN;
Avion = ARRAY[1..12] OF Comida;
Bote = ARRAY[1..12],[1..6] OF BOOLEAN;

VAR
Index,Count : CARDINAL;
Aux    : ArrayDef;
Aux2   : ArrayDef;
Aux3   : ARRAY[12..25] OF INTEGER;
Aviones : Avion;
Botes : Bote;

BEGIN
.
.
.
END Types.
```

3.5. Identación y Puntuación

Es necesario utilizar puntuación para decirle al compilador cuando un *statement* termina. Se utiliza *;* en las siguientes situaciones:

- Después de la declaración del programa
- Después de cada definición de constante
- Después de cada definición de variable
- Después de cada definición de tipo
- Después de cada *statement*

3.6. Comentarios en el códogo

Para comentar en *Modula-2* se inicia con (* y termina con *), lo anterior para comentarios de una sola línea y también para comentarios de varias lineas, pero no se pueden delimitar entre llaves {}, cosa que en *Pascal* si se puede.

3.7. Condiciones

Modula-2 usa la misma estructura que *Pascal* para verificar condiciones

- *IF*: Se utiliza para comprobar una sola condición o muy pocas.
- *CASE*: Se utiliza para casos de selección múltiple.

Cuadro 13. Sintaxis de un IF

```
MODULE if1;

FROM InOut
IMPORT WriteString;

VAR
Cont: CARDINAL;

BEGIN

Cont := 3;

IF Cont=4 THEN
WriteString('Cont = 4');
ELSIF Cont=3 THEN
WriteString('Cont = 3');
ELSE
WriteString('Cont no es 3 ni 4');

END;

END if1.
```

Luego del *IF* se indica la condición que se quiere comprobar, seguida por la palabra *THEN* y por los comandos a ejecutar si se cumple la condición. Al contrario de *Pascal* que si se esperaba una sola orden se debía de emplear el *BEGIN* y el *END* si eran varias ordenes. Ademas, se utiliza la condición *ELSE* para ejecutar los pasos si no se cumple la condición, pero si ocupamos comprobar otra nueva condición se puede usar *ELSIF*

Cuadro 14. Sintaxis de un Case

```
MODULE caseW;

FROM InOut
IMPORT WriteString;

VAR
num: CARDINAL;

BEGIN

num := 7;

CASE num OF
1,2: WriteString('num vale 1 o 2');   |
3: WriteString('num vale 3');         |
4..9: WriteString('num vale entre 4 y 9');
ELSE
WriteString('num no esta entre 1 y 9');
END;

END caseW.
```

La implementación del *CASE* es similar a la de *Pascal*, solo tiene una diferencia al terminar cada posible opción se emplea el símbolo |, (barra vertical).

3.8. Estatutos *WHILE*

Ira con un conjunto de lineas que deben de terminar con *END* (ligado al *WHILE*). Esto es algo que no sucede en *Pascal*.

Cuadro 15. Sintaxis de un While

```
MODULE bucle;

FROM InOut
IMPORT WriteString, WriteCard, WriteLn;

VAR
i: CARDINAL;

BEGIN

(* Contamos de 1 a 10 con WHILE *)

i := 1;
WHILE i <= 10 DO
(* Escribe un n mero entero no negativo *)
WriteCard(i, 4);
(* El 4 representa la anchura (espacios) *)
INC(i);
END;

WriteLn;

END bucle.
```

3.9. Estatutos *REPEAT*

Similar al estatuto *WHILE*, pero primero realiza los estatutos declarados en su bloque y luego pregunta la condición

Cuadro 16. Sintaxis de un REPEAT

```
MODULE bucle;

FROM InOut
IMPORT WriteString, WriteCard, WriteLn;

VAR
i: CARDINAL;

BEGIN

(* Contamos de 1 a 10 con REPEAT *)

i := 1;
REPEAT
WriteCard(i, 4);
INC(i);
UNTIL i>10;

WriteLn;

END bucle.
```

3.10. Estatutos *FOR*

Después del *FOR* ira el conjunto de lineas que terminan con *END*. Diferente a *Pascal* ya que después del *FOR* se puede indicar un único comando pero si se quiere que sean varios se deben de delimitar entre *BEGIN* y *END*.

Cuadro 17. Sintaxis de un FOR

```
MODULE bucle;

FROM InOut
IMPORT WriteString, WriteCard, WriteLn;

VAR
i: CARDINAL;

BEGIN

(* Contamos de 1 a 10 con FOR *)

FOR i := 1 TO 10 DO
WriteCard(i, 4);
END;

WriteLn;

END bucle.
```

Ademas se pueden contar con incrementos distintos a 1, por ejemplo si queremos contar de 4 en 4, debemos usar *FOR i := 1 TO 30 BY 3 DO ...* y para contar hacia atrás, seria *FOR i := 1 TO 30 BY -1 DO ....*

3.11. Estatutos *LOOP*

Repite indefinidamente esa parte del programa, que tiene que terminar con *END*, si queremos salir del bucle se debe de utilizar el comando *EXIT*. En el siguiente ejemplo se sale del bucle cuando el contador es mayor a 10.

Cuadro 18. Sintaxis de un LOOP

```
MODULE bucle;

FROM InOut
IMPORT WriteString, WriteCard, WriteLn;

VAR
i: CARDINAL;

BEGIN

(* Contamos de 1 a 10 con LOOP *)

i := 1;
LOOP
WriteCard(i, 4);
INC(i);
IF i>10 THEN
EXIT;
END;
END;
WriteLn;
Universidad Nacional de Educaci n a Distancia
END bucle.
```

3.12. Puntos Importantes[2]

- INC: Incrementa el valor de una variable de 1 en 1, pero también se puede definir en cuanto se quiere realizar cada salto de la siguiente manera *INC(b,4)*, así se aumentaría el valor de la variable *b* en 4 unidades.
- DEC: Decrementa el valor de una variable.
- WriteLn: Escribe un salto de linea.
- WriteCard: Escribe un *CARDINAL* y se le aplica la anchura para mostrarlo, si el numero ocupa menos de la anchura dada se rellenara con espacios en blanco por la izquierda pero si ocupa mas no se borrara el numero.

3.13. Procedures

Un *procedure* es un conjunto de sentencias, que se pueden llamar para hacer un trabajo en especial. El *procedure* no retorna nada y ademas no se termina con . si no que se termina con ;.

Cuadro 19. Sintaxis de un Procedure sin parametros

```
MODULE Proced1;

FROM InOut
IMPORT WriteString, WriteInt, WriteLn;

PROCEDURE Saludar;
BEGIN
WriteString('Hola');
WriteLn;
END Saludar;

BEGIN

Saludar;
WriteLn;

END Proced1.
```

La diferencia de un *procedure* en *Pascal* es que la palabra *procedure* debe de escribirse en mayúsculas y después del *END* se debe de repetir el nombre del procedimiento que termina. Para llamar un *procedure* se utiliza su nombre más los datos para satisfacer la lista de parámetros declarados y se puede llamar a un *procedure* dentro de otro.

Cuadro 20. Sintaxis de un Procedure con parametros

```
MODULE Proced2;

FROM InOut
```

2. Datos Obtenidos de *Programacin 1.*, Departamento de Lenguajes y Ciencias de la Computacin, Universidad de Málaga.

```
IMPORT WriteString, WriteInt, WriteLn;

VAR
cont : INTEGER;
cont2 : INTEGER;

PROCEDURE MostrarInfo(aux : INTEGER);
BEGIN
WriteString("El valor del param es  ");
WriteInt(aux,5);
WriteLn;
aux := 12;
END MostrarInfo;

BEGIN

FOR cont := 3 TO 5 DO
cont2 := cont;
MostrarInfo(cont2);
WriteString("El valor es  ");
WriteInt(cont2,5);
WriteLn;
END;

END Proced2.
```

3.14. Function Procedure

Cabe aclarar que funciones y *procedures* no son lo mismo, las funciones tienen valores de retorno mientras que los *procedures* no. En *Modula-2* no se usa la palabra reservada *Function* si no que es un *procedure* que devuelve un valor. *Modula-2* sabe que un *procedure* es una función ya que después de los parámetros se define el tipo del valor de retorno.

Cuadro 21. Sintaxis de una función

```
MODULE Func;

FROM Terminal2
IMPORT WriteString, WriteInt, WriteLn;

PROCEDURE SumaXCuatro
        (Num1, Num2 : INTEGER) : INTEGER;
BEGIN
RETURN(4*(Num1 + Num2));
END SumaXCuatro;

VAR
Perros, Gatos, Patas : INTEGER;

BEGIN
Perros := 4;
Gatos := 3;
Patas := SumaXCuatro(Perros,Gatos);
WriteString("En total hay ");
WriteInt(Patas,3);
WriteString(" patas.");
WriteLn;
END Func.
```

3.15. Recursión

Recursión es llamar un *procedure* en el mismo, es importante recordar utilizar una condición de parada en la recursión para evitar que esta se vuelva infinita. Se hace exactamente igual que en *Pascal* seguidamente se presenta un ejemplo del uso de la recursión:

Cuadro 22. Recursión en Modula-2

```
MODULE Recursion;

FROM Terminal2
IMPORT WriteString, WriteInt, WriteLn;

VAR
Cont : INTEGER;

PROCEDURE MostrarYRestar(Param : INTEGER);
BEGIN
WriteString("El valor es ");
WriteInt(Param,5);
WriteLn;
Param := Param - 1;
IF Param > 0 THEN
MostrarYRestar(Param);
END;
END MostrarYRestar;

BEGIN
Cont := 7;
MostrarYRestar(Cont);
END Recursion.
```

3.16. Operadores Booleanos

Operador	Significado
and	Las dos condiciones evaluadas tienen que ser verdad
and then	Igual que *and* solo que garantiza que se evalúen las expresiones en el orden dado
or	Al menos una de las condiciones debe ser verdad
or else	Igual que *or* solo que garantiza que se evalúen las expresiones en el orden dado
not	Niega el resultado (es unario)
xor	*or* exclusivo

3.17. Operadores relacionales

Es importante mencionar que la asignación en *Modula-2* es ":=". Estos operadores permiten comparar dos valores del mismo tipo. Se sabe que son binarios ya que poseen dos argumentos. Los operadores son los siguientes:

Operador	Significado
=	Igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
<> #	Distinto a

3.18. Operadores aritmeticos.

Operador	Operación	Operandos	Tipo Resultado
+	Suma	Integer o Real	Integer si los dos operandos son integer, real de otra forma
-	Resta	Integer o Real	Integer si los dos operandos son integer, real de otra forma
$*$	Multiplicación	Integer o Real	Integer si los dos operandos son integer, real de otra forma
/	División	Integer o Real	Real
div	División truncada	Integer	Integer
mod	Modulo	Integer	Integer

3.19. Prioridad de algunas operaciones

1. not
2. $*$, $/$, div, mod, and
3. $+$, $-$, or
4. $<$ $>$ $<=$ $>=$ $=$ $<>$ #

4. Características

- La principal característica es que en *Modula-2* se crean programas modulares, formado por diferentes módulos que se relacionan, de aquí viene su nombre.
- Se puede dar concurrencia que es multitarea entre distintas partes de un programa.
- *Modula-2* fue influido por *Pascal* y *Mesa*.

5. Características y Ventajas

- Introduce el concepto de modulo y encapsulación.
- Se facilita una interfaz llamada modulo de definición.
- Utilizado para la enseñanza de lenguajes estructurados.

6. Desventajas

- Popularidad: *Modula-2* no es un lenguaje tan popular como lo puede ser Java u otros, esto conduce a que librerías y otros aditivos sean escasos para solucionar cierto tipo de problemas.
- Imposibilita la modificación de estructuras de programación a cualquier persona que no tenga el código de su modulo de implementación.
- Incapacidad de declarar múltiples instancias de los módulos.

Referencias

[1] N. Wirth. 2002. *Pascal and its Successors. In Software Pioneers.* In M. Broy and E. Denert, Eds. Springer-Verlag.

[2] C. Pronk. 1997. *Standardized extensions to Modula-2.* International conference. ACM New York, NY, USA, 34-48. DOI=http://dl.acm.org/citation.cfm?id=270949&CFID=981704189&CFTOKEN=40472858

[3] N. Wirth. 1980. *Modula-2.* Research Collection. Vol. 1. DOI=https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/68683/eth-3135-01.pdf

[4] D. Pase. 1985. *System programming in Modula-2.* ACM New York, NY, USA, 49 - 53. DOI=http://dl.acm.org/citation.cfm?id=988299&CFID=981704189&CFTOKEN=40472858

[5] C, Somolinos. 2000. Fundamentos de programación con Modula 2. (1 edicin). Editorial Centro de Estudios Ramón Areces, S.A. p. 486.