Notación Algorítmica

A continuación se decribe la notación algorítmica Alpha.

1 Sintaxis Básica

En este documento se tomará las siguientes convenciones:

- En las definiciones de tipo, los símbolos que esten dentro de corchetes $\left[\ \right]$ se consideran opcionales.
- En una sola línea de código, se definen variables de un solo tipo de dato.
- Para una mejor legibilidad de los algoritmos, las variables del tipo **Integer** comienzan con la letra **i**, del tipo **Char** con la letra **c**, del tipo **Real** con la letra **r**, del tipo **Boolean** con la letra **b**, del tipo **Enum** con la letra **e**, del tipo **Pointer** con la letra **p**, del tipo **String** con la letra **s**, del tipo **Array** con la letra **a**, del tipo **Register** con la cadena **rg** y, del tipo **File** con la letra **f**. Si es un tipo definido (i.e. **Type**), la variable comienza con la letra **t**.
- Los identificadores de las variables del tipo constante son en mayúsculas.
- El nombre de los procedimientos y funciones comienzan con letra mayúscula
- Los atributos de las clases tienen como prefijo la cadena **m_** para indicar que son variables miembro del la clase.

1.1 Tipo de Dato Simple

Tipo Caracter - Char

- Conjunto de valores: 'A' - 'Z' $\cup$ 'a' - 'z' $\cup$ '0' - '9' $\cup$ conjunto de caracteres de la tabla ASCII
- Operadores: relacionales ($==$, $\neq$, $>$, $\geq$, $<$, $\leq$)

Tipo Lógico - Boolean

- Conjunto de valores: true, false
- Operadores: relacionales de álgebra booleana (*and*, *or*, *not*, $==$, $\neq$)

Tipo Entero - Integer

- Conjunto de valores: $S \subset \mathbb{Z}$
- Operadores: aritméticos ($+$, $-$, $*$, $\hat{}$, *div*, *mod*) y relacionales ($==$, $\neq$, $>$, $\geq$, $<$, $\leq$)

Tipo Real - Real

- Conjunto de valores: $S \subset \mathbb{R}$
- Operadores: aritméticos ($+$, $-$, $*$, $\hat{}$, /) y relacionales ($==$, $\neq$, $>$, $\geq$, $<$, $\leq$)

Tipo Enumerado - Enum

- Definición de tipo: **Enum** $< identificador > = [< id_1 >, < id_2 >, ..., < id_n >] \left[\ ;\ \right]$ ó
  Enum $< identificador > = [\ \ < id_1 >,$
  $< id_2 >,$
  ...,
  $< id_n > \left[\ ;\ \right]$
- Conjunto de valores: Subconjunto de valores enteros especificados por el intervalo $[1, n]$, donde n es el número de identificadores del tipo **Enum**.
- Operadores: Heredados del tipo **Integer**

Tipo Apuntador - Pointer

- Definición de tipo: $< tipo > \star < identificador > \left[\ ;\ \right]$
- Conjunto de valores: Direcciones de memoria donde se almacenen variables del tipo $< tipo >$
- Operadores: Referenciación (ref) y derreferenciación ($\star$)

1.2 Operaciones Básicas

Las operaciones simples del lenguaje se definen como la declaración de variables, asignación de valores a variables, la lectura estándar y escritura estándar. Desde este punto en adelante, cuando se refiera a $< tipo >$ este representa cualquier tipo de dato válido en la notación.

1.2.1 Declaración de variables

La declaración de una variable de nombre $< identificador >$ del tipo de dato $< tipo >$ se declara como:

```
<tipo> <identificador> [ ; ]
```

Del mismo modo, es posible definir múltiples variables como una lista de identificadores en su declaración:

```
<tipo> <identificador_1, identificador_2, ..., identificador_k> [ ; ]
```

Nótese que la presencia del símbolo ; al final de la declaración es opcional. Si se aplica dicho operador, se considera como de secuenciamiento, es decir, permite declarar más instrucciones en una misma línea. Algunos ejemplos de declaraciones válidas:

```
Char c1, c2
Boolean bIsValid
Real rX, rY;    Boolean bUseful;
Integer iDay, iMonth, iYear;
```

Se recomienda emplear una instrucción por línea solo para hacer más legible el código a escribir. Para los ejemplos presentados en este documento no se empleará el uso del símbolo ; al final de cada línea.

1.2.2 Asignación Simple

La asignación simple permite guardar el valor de un valor en una variable declarada. Para la declaración de un $< valor >$ a una variable $< identificador >$ se realiza de las siguientes formas:

```
<identificador> = <valor> [ ; ]
<identificador> = <identificador> [ ; ]
```

Se emplea el operador = para la asignación. La asignación no solamente permite darle un valor a las variables, sino también una expresión. Algunos ejemplos válidos:

```
Real rTake = 3.07
Integer iValue = 5
Boolean bStatus = iValue > 0
Integer iCopy = iValue
```

En el caso de las expresiones, ésta se intenta evaluar de izquierda a derecha, mientras sea posible. Si existen diversos operadores, entonces se debe aplicar prioridad de operadores. La prioridad de operadores define un orden de ejecución de los operadores dentro de una expresión. Si en la expresión existen operadores con el mismo nivel de prioridad, se evalúa de izquierda a derecha.

Prioridad de Operadores

De selección . [] () $*$ (derreferenciado)

Aritméticos-1 $\hat{}$ $-$ (unario) $*$ / *div mod*

Aritméticos-2 $*$ / *div mod*

Aritméticos-3 $+$ $-$

Relacionales-1 $> < \geq \leq$

Relacionales-2 $== \neq$

Lógicos *not and or*

Asignación =

Un ejemplo se la prioridad de operadores se puede observar a continuación:

```
Boolean bValue
bValue = -4^2 < 8.05 + 6 * 3 and not 3 == 8
```

Para obtener el valor de $bValue$ primero se evalua el símbolo $-$ unario en el 4 y luego $-4\hat{}2$ dando un valor de -16. Luego, se evalúa $6 * 3$ y posterior se suma el valor de 8.05 resultando el valor de 26.05. Después, se realiza la comparación $-16 < 26.05$ dando el valor de $true$. La próxima evaluación es $3 == 8$ resultando en $false$, y not de $false = true$. Finalmente, la evaluación final es $true\ and\ true$ en la asignación de $bValue$.

El mecanismo para violar la prioridad existente es el uso de paréntesis ().

1.2.3 Declaración de constantes

Una constante $< identificador >$ se define de acuerdo a su $< tipo >$ y a su $< valorDeclarado >$ como:

```
Const <tipo> <identificador> = <valorDeclarado> [ ; ]
```

El $< valorDeclarado >$ debe pertenecer al conjunto definido para el $< tipo >$ de forma explícita. Ejemplos de declaraciones de constantes se muestran a continuación:

```
Const Integer MAX_MONTHS = 12
Const String COUNTRY = "Venezuela"
Const Real PI = 3.14159
```

1.2.4 Entrada y Salida Simple

La entrada o lectura simple permite obtener desde el dispositivo de entrada estándar definido (e.g. teclado) y asignarlo a una ó más variables del mismo tipo definidas como $< identificador - k >$ previamente declaradas. Su forma sentencial es:

```
Read (<identificador-1>, <identificador-2>, ..., <identificador-k>) [ ; ]
```

La salida o escritura simple permite mostrar los resultados de una expresión, identificador o valor por el dispositivo de salida estándar (e.g. monitor, impresora, etc.). Se emplea de las siguientes formas:

```
Print (<identificador-1>, <identificador-2>, ..., <identificador-k>) [ ; ] //lista de identificadores
Print (<valor>) [ ; ] //un solo valor
```

Un ejemplo de salidas válidas en la notación algorítmica es:

```
Integer iQuantity = 8
Print (6.6)
Print (iQuantity)
Print ("El valor de iQuantity es: " + iQuantity)  //empleando el operador de concatenacion de String
```

A continuación se muestra un ejemplo de asignación de valores para cada uno de los tipos elementales mostrados previamente.

Código 1: *Ejemplo de definiciones.*

```
Integer iMonth = 6
Char cGender = 'g'
Boolean bFlag = false
Real rTaxes = 12.5
Enum eDays = [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]
```

1.2.5 Comentarios

Los comentarios ocupan solo una línea y se identifican por el símbolo //. Por ejemplo:

```
Integer iMonth = 6    //esto es un comentario
// Boolean bAmIalive = false esta linea no se ejecutara
// Para realizar comentarios de mas de una linea,
// se debe emplear el simbolo // en cada linea
```

1.3 Tipo de Dato Compuesto

Dentro de la notación se definen las cadenas (**String**), los arreglos (**Array**), registros (**Register**) y archivos (**File**).

1.3.1 Tipo Cadena - String

El tipo String se puede definir como una secuencia de caracteres.

- Definición de tipo: **String** $< identificador >$ [;]
- Conjunto de valores: Secuencia de valores del tipo **Char** sin límite definido
- Operadores: concatenación (+), relacionales (==, $\neq$), de acceso ([$<$ *pos* $>$]), y de construcción de substring ([$< li >$ .. $< ls >$]).

El operador de acceso permite obtener un valor del tipo **Char** de un tipo **String**. El valor de $< pos >$ corresponde a un valor del tipo **Integer** en el rango $[1, N]$ donde N representa la longitud del **String**. Un ejemplo se muestra a continuación:

```
String sName = "Springfield" //contiene 11 posiciones
Integer iPos = 4
Char c = sName[iPos]  //es equivalente a sName[4]
Print (c)  //el resultado es 'i'
```

Del mismo modo, es posible construir un nuevo tipo **String** de longitud menor o igual al tipo **String** original. Para ello, se emplea el operador de acceso pero en vez de una posición, es el límite inferior seguido de dos puntos continuos y el límite superior (ambos dentro del rango de la longitud del String). Un ejemplo de ello se muestra a continuación:

```
String sText = "Hello World!!"
String sNewString = sText[3..5]
Print (sNewString) //la salida es "llo"
```

1.3.2 Tipo Arreglo - Array

Para el arreglo se realiza la definición del tipo de dato, la operación selectora que permite obtener un valor del arreglo y la operación constructora para inicializar los valores de un arreglo.

- Definición de tipo: **Array** $< identificador >$ **of** $< tipo >$ $[L_i..L_s]$ [;]
- Selector: $< identificador >$[$< pos >$], donde *pos* está en el rango $[L_i, L_s]$
- Constructor: **Array** $< identificador >$ **of** $< tipo >$ [ ] = **{**$valor_1$, $valor_2$, ..., $valor_k$**}** [;]

Por ejemplo, para imprimir el valor de la 2da posición de un arreglo de nombre $aValues$ que contiene 6 valores del tipo **Integer**, se hará de la siguiente forma:

```
Array aValues of Integer [] = {1, 2, 3, 4, 5, 6}
Integer iValue = aValues[2]
Print (iValue)  //es equivalente a Print (aValues[2])
```

El tipo **Array** soporta construir estructuras que sean k-dimensionales. Por ejemplo, para $k = 2$ se define como:

Array $< identificador >$ **of** $< tipo >$ $[L_i^1..L_s^1][L_i^2..L_s^2]$ [;]

donde L_i^1 y L_s^1 definen los límites inferior y superior de la primera dimensión, y L_i^2 y L_s^2 de la segunda dimensión.

Igualmente, este concepto se puede llevar a k-pares de $[L_i^k$ .. $L_s^k]$ para la definición de un arreglo k-dimensional.

1.3.3 Tipo Registro - Register

En la definición del tipo de dato **Register** se muestra la forma de su definición, el operador de acceso y el conjunto de valores.

- Definición de tipo:
  Register $< identificador >$
  $< tipoDato_1 >$ $< identificador_1 >$ [;]
  $< tipoDato_2 >$ $< identificador_2 >$ [;]
  ...
  $< tipoDato_n >$ $< identificador_n >$ [;]
  end

- Selector: El operador **.** para tener acceso a un campo del registro. La asignación = seguida de todos los literales asociados a cada campo, separados por el símbolo **,** entre llaves { }
- Conjunto de Valores: Los asociados a cada tipo de dato de los campos

Un ejemplo de definición de un tipo **Register** que almacena los datos de una persona podría ser:

```
Register rgPerson
  String sName
  Integer iId
  Char cGender
  String sAddress
end
```

Al mismo tiempo, para asignarle valores a la variable *rPerson* se puede realizar con el operador selector . de la siguiente forma:

```
rgPerson.sName = "Homer Simpson"
rgPerson.iId = 911
rgPerson.cGender = 'M'
rgPerson.sAddress = "742 Evergreen Terrace"
```

Y también es posible con el operador de asignación = con los valores literales como una lista:

```
rgPerson = {"Homer Simpson", 911, 'M', "742 Evergreen Terrace"}
```

1.3.4 Tipo Archivo - File

Un tipo de dato **File** representa a un archivo ó fichero. Este tipo de dato se construye como una clase, y sus variables definidas como objetos. A continuación se muestra su definición, y las funciones asociadas a ésta.

La declaración de un archivo secuencial es:

File $< objetoF >$ [;]

Las operaciones básicas de un tipo **File** son:

Abrir Archivo La definición para abrir un archivo en una variable $< identificadorF >$ del tipo **File**:
Boolean $< identificador > = < objetoF >.OpenFile\ (< nombreArchivo >, < parametros >)$ [;]
donde $< nombreArchivo >$ es un tipo String que representa el nombre del archivo a abrir y, $< parametros >$ se refiere a la forma de operar con el archivo y puede tomar los siguientes valores:

Create : Indica que se creará el archivo

Write : Indica que el archivo se abre de solo escritura

Read : Indica que el archivo se abre de solo lectura

Text : Indica que el archivo a abrir es de formato texto

Binary : Indica que el archivo a abrir es de formato binario

Append : Indica que el archivo se abre de escritura pero se añade al final de archivo

Es posible combinar el campo $< parametro >$ con el operador lógico *and*. Una sentencia válida es:

```
File fMyFile
Boolean bResult = fMyFile.OpenFile("photo.jpg", Read and Binary)
```

La variable $< identificador >$ es una variable del tipo **Boolean** que obtiene el resultado de la operación *OpenFile*, si fue exitosa toma el valor de **true** y en caso contrario **false**. Esta condición aplica para las otras funciones del tipo **File**.

Cerrar Archivo Para cerrar o finalizar de operar sobre un archivo se hace de la siguiente forma:
$Boolean < identificador >= < objetoF >.CloseFile\ (\ )$ [;]

Fin de Archivo La instrucción para indicar la finalización de archivo es:
$Boolean < identificador >= < objetoF >.EndFile\ (\ )$ [;]

Leer y Escribir del Archivo Para la lectura del contenido del archivo se debe emplear una variable $< identificador >$ que almacene los valores leídos. La definición es de la siguiente forma:
$Boolean < identificador >= < objetoF > .ReadFile\ (< identificador >)\ [\ ;\ ]$

De forma similar, la escritura en un archivo se realiza como:
$Boolean < identificador >= < objetoF > .WriteFile\ (< identificador >)\ [\ ;\ ]$

1.4 Tipo Definido - Type

La notación permite la definición de nuevos tipos de dato. Para ello, se debe colocar previo a la definición de una variable la palabra **Type**. De esta forma, una declaración válida quedará como

```
Type Array aWeek of Boolean [1..7]
```

Un nuevo tipo de dato llamado *aWeek* ha sido creado que representa a un arreglo de 7 posiciones del tipo **Boolean**, siendo posible crear variables de este tipo:

```
aWeek tLaboral, tHolidays
```

1.5 Tipo Apuntador - Pointer

Se define de la siguiente forma:

```
Integer* pToInteger1, pToInteger2
Real* pPointer2Real
```

1.5.1 Operador Referenciación

```
Integer* pToInteger
Integer iValue = 9
pToInteger = ref iValue
```

1.5.2 Operador Derreferenciación

```
(*pToInteger) = (*pToInteger) + iValue
Print ((*pToInteger)) //imprime 18
pToInteger = NIL      //equivalente a pToInteger = NIL
```

1.5.3 Asignación Dinámica

Se utiliza la constante **NIL**,y los operadores **new** y **delete** para la creación dinámica y para eliminar un apuntador.

```
Integer * pMyPointer = new Integer;  //reserva el espacio de memoria para un tipo Integer
pMyPointer = 6
Integer * pMyArray = new Integer[1..8];  //reserva el espacio de memoria de dimension 8
Integer iResult = (*pMyPointer) + pMyArray[5]
pMyPointer = NIL
pMyPointer = new Integer[1..8]
delete pMyPointer
delete pMyArray
```

1.6 Estructuras de Control

Las estructuras de control permiten componer operaciones basadas en un conjunto de condiciones que hacen que se ejecute de forma excluyente un código. La notación define el condicional ó selección simple, condicional doble y selección múltiple.

1.6.1 Condicional Simple

Dada una condición $< condicionB >$ que representa a una expresión que retorna un valor del tipo **Boolean**, se puede definir un condicional simple como:

if $< condicionB >$ **then**
$< instruccion_1 >$
$< instruccion_2 >$
...
$< instruccion_n >$
end

Dentro del cuerpo del condicional puede haber 1 ó más instrucciones.

1.6.2 Condicional Doble

Dada una condición $< condicionB >$ que representa a una expresión que retorna un valor del tipo **Boolean**, se puede definir un condicional doble como:
if $< condicionB >$ **then**
$< instruccion_1^A >$
$< instruccion_2^A >$
...
$< instruccion_n^A >$
else
$< instruccion_1^B >$
$< instruccion_2^B >$
...
$< instruccion_m^B >$
end

Si $< condicionB >$ toma el valor de **true** entonces se ejecutan el conjunto de $< instruccion_i^A >$, en caso contrario el conjunto de $< instruccion_i^B >$. Como parte de las instrucciones puede estar uno ó más condicionales simples o dobles.

Cuando luego de la sentencia else, existe otro condiciones (simple o doble) es posible emplear la sentencia **elseif**. Su utilidad radica en hacer más legible el código escrito la notación. Su sintaxis es:

if $< condicion_1 >$ **then**
$< instruccion_1^A >$
$< instruccion_2^A >$
...
$< instruccion_n^A >$
elseif $< condicion_2 >$ **then**
$< instruccion_1^B >$
$< instruccion_2^B >$
...
$< instruccion_m^B >$
end

1.6.3 Selección Múltiple

Dado un conjunto de condiciones que son excluyentes entre sí, se puede definir la estructura de selección múltiple(**select**) como:
select
$< condicion_1 >: < instruccion_1 >$
$< condicion_2 >: < instruccion_2 >$
...
$< condicion_n >: < instruccion_n >$
end

```
Integer iAge
Print ("Ingrese su edad: ")
Read (iAge)
select
iAge <= 0: Print ("Ud. no ha nacido")
iAge > 1 and iAge < 15: Print ("Ud. es un infante")
iAge >= 15 and iAge < 60: Print ("Ud. es un adulto")
iAge >= 60 and iAge < 120: Print ("Ud. es un anciano")
iAge >= 120: Print ("Ud. es un inmortal")
end
```

Cabe destacar que la estructura **select** solo es válido si contempla una partición de las condiciones posibles, es decir, solo satisface una de las condiciones $< condicion_k >$. La estructura **select** se basa en la partición del rango de una variable y la verificación de las condiciones solo se hacen con un valores literales.

1.7 Estructuras Iterativas

Básicamente, las estructuras iterativas permiten ejecutar un conjunto de instrucciones de forma condicionada. La notación se definen 3 tipos de estructuras iterativas: **while**, **do-while** y **for**. Es importante destacar que la mayoría de los lenguajes de programación implementan estos tres tipos, pero existen otros tipos como la sentencia *foreach*.

1.7.1 Mientras - While

Dada una $< condicion >$, la estructura iterativa **while** se define como:
while $< condicion >$ **do**
$< instruccion_1 >$
$< instruccion_2 >$
...
$< instruccion_n >$
end

```
Integer iFactorial = 1, iNumber = 8
while iNumber > 1 do
  iFactorial = iFactorial * iNumber
  iNumber = iNumber - 1
end
Print ("8! = " + iFactorial)
```

1.7.2 Hacer-Mientras - Do-while

do
$< instruccion_1 >$
$< instruccion_2 >$
...
$< instruccion_n >$
while $< condicion >$ [;]

```
Integer iFactorial = 1, iNumber = 8
do
  iFactorial = iFactorial * iNumber
  iNumber = iNumber - 1
while iNumber > 1
Print ("8! = " + iFactorial)
```

1.7.3 Para - For

for $< identificador >$ = $< vInicial >$ **to** $< vFinal >$ [**step** $< incremento >$] **do**
$< instruccion_1 >$
$< instruccion_2 >$
...
$< instruccion_n >$
end

```
Integer iFactorial = 1
for Integer iNumber = 8 to 1 step -1 do
  iFactorial = iFactorial * iNumber
end
Print ("8! = " + iFactorial)
```

1.7.4 Para cada - Foreach

foreach $< identificador >$ **in** $< Container >$
$< instruccion_1 >$
$< instruccion_2 >$
...
$< instruccion_n >$
end

```
Array aValues of Integer [] = {2,3,4,5,6,7,8}
foreach iNumber in aValues
  iFactorial = iFactorial * iNumber
end
Print ("8!=" + iFactorial)
```

1.8 Procedimientos

1.8.1 Acción - Void

void < *identificador* > **(** < *parametros* > **)**
 < *definiciones* >
 < *instrucciones* >
end

```
void Factorial (Integer iN)
  Integer iFactorial = 1
  while iN > 1
    iFactorial = iFactorial * iN
    iN = iN - 1
  end
  Print (iN + "! = " + iFactorial)
end
```

El pase de parámetros puede ser por valor o por referencia. Por defecto, los valores son pasados por valor. Para pasar un parámetro por referencia debe ser solamente un identificador y anteponer la palabra **ref**.

1.8.2 Función - Function

function < *identificador* > **(** < *parametros* > **) :** < *tipo* >
 < *definiciones* >
 < *instrucciones* >
 return <*var_de_tipo*> [;]
end

```
function Factorial (Integer iN) : Integer
  Integer iFactorial = 1
  while iN > 1
    iFactorial = iFactorial * iN
    iN = iN - 1
  end
  return iFactorial
end
```

```
for Integer iNumber = 1 to 10 do
  Print (iNumber + "! = " + Factorial(iNumber))
end
```

2 Orientado a Objetos

```
class CRectangle
private:
  //attributes
  Real m_rWidth, m_rHeight
public:
  //constructors
  Constructor CRectangle ()
    m_rWidth = 0.0
    m_rHeight = 0.0
  end

  Constructor CRectangle (Real rWidth, rHeight)
    m_rWidth = rWidth
    m_rHeight = rHeight
  end
```

```
  //methods
  function GetArea () : Real
    return m_rWidth * m_rHeight
  end

  function GetPerimeter () : Real
    return 2 * (m_rWidth + m_rHeight)
  end

  void PrintInfo()
    Print ("Area =" + GetArea())
    Print ("Perimeter =" + GetPerimeter())
  end
end
```

** Este documento está en constante revisión

esmitt ramírez / 2015