Introducción al AlgoLang

Matias Gobbi

27 de octubre de 2019

Fundamentos

Antes de comenzar propiamente con la definición formal e implementación de nuestro intérprete, lo correcto es dar una introducción al lenguaje. Por lo tanto, en la siguiente sección nos dedicaremos a dar el significado intuitivo de las distintas construcciones que ofrece el AlgoLang. Detallaremos de forma general distintos aspectos sintácticos y semánticos del mismo.

1. Introducción

El AlgoLang es un pseudocódigo diseñado para el desarrollo de la materia *Algoritmos y Estructura de Datos II*. Por lo tanto, su uso está orientado para la enseñanza de los contenidos de la asignatura. Los ejes principales de la materia consisten en el análisis de algoritmos, la definición de estructuras de datos y la comprensión de algoritmos avanzados.

El objetivo del pseudocódigo es poder introducir a los estudiantes a nuevos conceptos, y fomentar buenas técnicas de programación. Se utiliza para describir de forma informal principios operacionales de los distintos algoritmos estudiados en la materia. Típicamente, se omiten detalles esenciales para la implementación de los programas para favorecer el entendimiento de los mismos. No existe ningún estándar para la sintaxis o semántica de AlgoLang, por lo que un programa en este lenguaje no es un programa ejecutable.

Este lenguaje (o más bien, pseudocódigo) está basado en el lenguaje de programación imperativo Pascal. El mismo fue diseñado por Niklaus Wirth cerca de 1970. Algunos elementos básicos que comparten son: 1. Una sintaxis simple, verbosa pero fácil de leer. 2. Un tipado fuerte para las expresiones. 3. Un formato estructurado del código.

En el resto de la sección nos referiremos a AlgoLang como un lenguaje definido, a pesar de ser simplemente un pseudocódigo. Nuestro objetivo final, es poder realizar una implementación tanto del lenguaje como de un interprete para el mismo.

2. Características Principales

El lenguaje está diseñado para enseñar conceptos fundamentales de forma clara y natural. No existe ninguna implementación concreta del mismo, y tampoco hay una definición formal de ninguna de sus componentes. Todo esto tiene ciertas consecuencias importantes.

La más inmediata es que el lenguaje es flexible en cuanto al nivel de abstracción que maneja, lo cual resulta conveniente a lo largo de la materia. En la cátedra se suelen omitir los detalles de implementación de distintas funciones y procedimientos para poder concentrarse en los conceptos propios de algoritmos. El lenguaje permite trabajar con estructuras como los arreglos para definir algoritmos de ordenación, o con listas enlazadas cuando se hace uso de la memoria de forma dinámica. Pero también permite trabajar con grafos o conjuntos para los algoritmos más abstractos. Obviamente, esto facilita la enseñanza de la materia pero posee una complicación fundamental para el desarrollo de un intérprete, la ambigüedad y falta de especificación.

Otra característica importante del lenguaje es el tipado fuerte. Toda expresión posee un tipo, y la misma debe ser respetada. Por ejemplo, si una variable x es de tipo *int* la misma no podrá ser usada como *bool*. Al mismo tiempo, el lenguaje ofrece varias construcciones polimórficas, y operadores sobrecargados, que funcionan para un rango de tipos distintos. En primera instancia, no hay conversiones de tipo en el lenguaje.

Debido a que no podemos ejecutar un programa de AlgoLang, ya que no existe ningún intérprete ni compilador para este lenguaje, el mismo no ofrece ciertas construcciones que en la actualidad son básicas para cualquier lenguaje de programación. A continuación enumeramos algunas: 1. Canales de input y output, ya sea con archivos o mediante interacción del usuario. 2. Manejo de errores y excepciones, no hay ningún mecanismo establecido para tratar estas situaciones. 3. Creación de módulos, la división de código no resulta una necesidad para el desarrollo de la materia.

3. Tokens

Comencemos propiamente con la descripción del lenguaje. Para la construcción del código, los bloques léxicos básicos se denomina tokens. Se podrían definir como las palabras del lenguaje. Los caracteres ingresados por el programador se combinan en tokens de acuerdo a las reglas del lenguaje de programación. Hay distintas clases de tokens en AlgoLang:

1. **palabras reservadas** son las que poseen un significado fijo en el lenguaje. No pueden ser modificadas o redefinidas.

2. **identificadores** son los nombres que el programador define para las variables, funciones, tipos, entre otras. Pueden ser reusadas, y están sujetas a las reglas de *scope*.

3. **operadores** son los símbolos para las operaciones. Entre los mismos se encuentran los aritméticos, los booleanos, entre otros. Se utilizan en las expresiones.

4. **separadores** comúnmente son los espacios en blanco. También pueden ser las tabulaciones o saltos de línea. No tienen mayor uso que mantener el código prolijo.

5. **constantes** se utilizan para denotar valores específicos en el código. Ejemplos de estos son los valores numéricos, los booleanos, entre otros.

A lo largo del desarrollo de esta introducción, iremos haciendo mención de cada uno de los mismos a medida que sea necesario. En el siguiente ejemplo, ilustramos algunos de los conceptos introducidos.

```
proc insert (in/out A: array [1..n] of T, in i: nat)
  var j: nat
  j := i
  do j > 1 ∧ A[j] < A[j-1] →
    swap(A, j - 1, j)
    j := j - 1
  od
end proc
```

Algunos ejemplos de palabras claves son *proc*, *in*, y *var*, que sus significados serán explicados más adelante. Los identificadores pueden ser *A*, *j*, y *swap*, los cuales representan parámetros, variables y procedimientos respectivamente. Los operadores utilizados son $>$, $\wedge$, $<$, y $-$. Y la única constante es el número *1*.

Un detalle importante en el lenguaje es que AlgoLang es *case sensitive*. Esto significa que el mismo distingue entre un token en minúsculas, de uno en mayúsculas. Debido a esto se pueden definir múltiples identificadores con el mismo nombre, pero que se diferencien por el uso de las mayúsculas o minúsculas.

3.1. Comentarios

Los comentarios son piezas de código que son descartados por el intérprete. Solo existen para el beneficio del programador, para poder explicar ciertas piezas del código. En el lenguaje solo se permiten realizar comentarios en una línea utilizando llaves, como en el siguiente ejemplo.

```
{ Este procedimiento inicializa un arreglo }
```

4. Tipos

Un tipo esencialmente, define el conjunto de valores que puede tomar una variable. En el lenguaje hay tipos básicos, tipos estructurados e incluso tipos definidos por el programador.

4.1. Tipos Básicos

Un tipo básico comprende un conjunto ordenable de valores. Existe un token para cada valor del conjunto. Poseen operaciones propias que los manipulan. En el lenguaje existen varios tipos básicos nativos.

4.1.1. Tipos Numéricos

Los tipos numéricos son los siguientes: **nat** , **int** y **real** . Estos representan los conjuntos de los números naturales, números enteros y números reales, respectivamente. En el lenguaje los números se especifican en notación decimal. A diferencia de su definición matemática, estos conjuntos están acotados. En el lenguaje se encuentran definidas las constantes $-\infty$ y $+\infty$, que representan los límites inferior y superior de los conjuntos de enteros y reales. Para el conjunto de los naturales, las cotas comprenden el *0* y el $+\infty$.

Asumiendo que n y m son expresiones numéricas del mismo tipo. Los operadores aritméticos que se ofrecen son los siguientes. Los mismos evalúan a un valor numérico.

Operador	Operación	Operandos
n + m	Suma	Numéricos
− n	Resta Unaria	Enteros/Reales
n − m	Resta Binaria	Numéricos
n ∗ m	Multiplicación	Numéricos
n / m	División Entera	Naturales/Enteros
n / m	División Exacta	Reales
n % m	Módulo	Naturales/Enteros
max(n, m)	Máximo	Numéricos
min(n, m)	Mínimo	Numéricos

Los operadores de comparación están definidos para una gran variedad de tipos, por lo que no se limitan solo a expresiones numéricas. De todas formas, listaremos los mismos en esta sección. Los operadores de comparación evalúan a un valor booleano.

Operador	Operación
n = m	Igualdad
n < m	Menor
n > m	Mayor
n ≤ m	Menor o Igual
n ≥ m	Mayor o Igual
n ≠ m	No Igualdad

Notar que los operadores para números están sobrecargados. Esto significa que las operaciones numéricas están definidas para naturales, enteros y reales. Por ejemplo, la suma se puede utilizar tanto para sumar naturales entre sí, como con reales o enteros. Lo que en el lenguaje no está especificado es si se permite utilizar operadores con operandos de distintos tipos. Por ejemplo, si se permite

sumar naturales con reales. Esta es una decisión que se tendrá que tomar una vez que se implemente el lenguaje.

4.1.2. Tipos Booleanos

El tipo **bool** representa al conjunto de valores booleanos. Puede adoptar los valores de verdad *True* y *False*. También se ofrecen las operaciones clásicas para manejo de booleanos. Asumiendo que b y q son expresiones de tipo booleano, los operadores son los siguientes.

Operador	Operación
$\neg$ b	Negación
b $\vee$ q	Disyunción
b $\wedge$ q	Conjunción

Un detalle importante en estas operaciones es el uso de la llamada evaluación por *short-circuit* (correspondiente a un modo de evaluación *lazy*). La misma determina que el segundo argumento de una operación binaria será evaluado solo si el valor del primer argumento no puede determinar el valor de la expresión completa.

4.1.3. Tipos Caracteres

El tipo **char** representa al conjunto de caracteres. Los valores que puede adoptar se representan encerrados entre comillas simples, de la forma *'a'*. Por sí solo, este tipo de datos no presenta mucha utilidad en el lenguaje y quedará relegado en un segundo plano. Inicialmente, las únicas operaciones que se pueden realizar entre caracteres son las comparativas.

En el siguiente ejemplo se detalla una función que utiliza los tipos básicos recientemente introducidos. La misma determina si un arreglo de caracteres posee llaves balanceadas, utilizando un contador entero.

```
fun balanceados (A: array [1..n] of char) ret b: bool
  var count: int
  count := 0
  b := true
  for i := 1 to n do
    if A[i] = '[' then
      count := count + 1
    else if A[i] = ']' ∧ count ≠ 0 then
      count := count - 1
    else if A[i] = ']' ∧ count = 0 then
      b := false
    fi
  od
  b := b ∧ count = 0
end fun
```

4.1.4. Tipos Enumerados

Los tipos enumerados representan uno de los tipos que puede ser definido por el programador en el lenguaje. Para los mismos, se enumeran todos los valores posibles, entre paréntesis, que puede adoptar este tipo de dato. Los valores serán especificados utilizando identificadores. Las únicas operaciones disponibles para estos tipos de datos son las comparativas.

```
type dia = (dom, lun, mar, mie, jue, vie, sab)
```

En el ejemplo anterior se define un nuevo tipo enumerado. Se utiliza el identificador *dia* para denotar al tipo. Los valores que puede adoptar el mismo están representados por los siete identificadores entre paréntesis.

4.2. Tipos Estructurados

Un tipo estructurado es caracterizado por el tipo de sus componentes y por su método de estructuración. En el lenguaje hay varios tipos estructurados. Al igual que los tipos básicos, también hay operaciones específicas definidas para los mismos.

4.2.1. Tipo Arreglo

Un tipo **array** es una estructura que consiste de una cantidad fija de componentes del mismo tipo. Los elementos de un arreglo son designados por índices. Los mismos deben representar un conjunto de valores enumerables. Para definir un arreglo se necesita especificar el tipo de los componentes y los índices de los mismos.

```
var A: array [1..4] of int
A[1] := 1
A[2] := 3
A[3] := 6
A[4] := 10
```

En este ejemplo, creamos un arreglo A que estará definido para los índices *1*, *2*, *3*, y *4*, y almacenará en cada posición valores de tipo entero. Luego, en cada posición i del arreglo se guardará la sumatoria de los primeros i números naturales. Notar que el siguiente programa es equivalente al anterior, donde utilizamos el estado del arreglo para el cómputo del mismo.

```
var A: array [1..4] of int
A[1] := 1
A[2] := A[1] + 2
A[3] := A[2] + 3
A[4] := A[3] + 4
```

La declaración de índices de arreglos no está solamente limitada a números. De hecho, todo los tipos que se puedan enumerar pueden ser utilizados como índices de arreglos.

```
var B: array ['a'..'z'] of bool
var C: array [-4..12] of char
var D: array [dom..sab] of real
B['g'] := True
C[-2] := 'v'
D[vie] := 3.14
```

En el ejemplo se declaran tres arreglos distintos. Los índices del arreglo de booleanos B son todos los caracteres ordenados alfabéticamente desde la letra *a* hasta la *z*. El arreglo de caracteres C tendrá índices que irán desde el número *-4* hasta el *12*. Los índices del arreglo de reales D comprenden los valores del tipo enumerado *dia*.

Hay una observación importante que hay que rescatar sobre los índices de un arreglo. Si el conjunto de índices de un arreglo es vacío, en el caso que el límite inferior sea mayor al límite superior, entonces el mismo tendrá longitud cero. Esto sería similar a tener un tipo vacío. Esta situación no tendría ninguna utilidad práctica.

Por último, los arreglos en AlgoLang no están limitados a una dimensión. En particular, uno puede crear arreglos multidimensionales simplemente especificando los rangos de las dimensiones necesarias al momento de declarar un nuevo arreglo.

```
var E: array [1..5, 1..5] of int
var F: array [1..3, 'a'..'e', 10..20] of real
E[1, 1] := 4
F[2, 'b', 15] := 4.0
```

El arreglo E posee dos dimensiones, ambas indexadas desde el número *1* al *5*. Para la asignación de valores del arreglo simplemente se separan con comas las distintas coordenadas que se quieren acceder o actualizar. El arreglo F, en cambio, tiene 3 dimensiones. La primera está indexada del *1* al *3*, la segunda por los caracteres de la *a* a la *e*, y la última, va del *10* al *20*.

4.2.2. Tipo Lista

Un tipo **list** es una estructura compuesta por una cantidad variable de componentes del mismo tipo. Una lista es similar a un arreglo, salvo por algunas diferencias fundamentales: 1. La longitud de una lista no está predefinida. Esto significa que el tamaño de la misma puede variar a lo largo de la ejecución de un programa. 2. Los valores de una lista no necesariamente se alojan en espacios contiguos de memoria. Este detalle es propio de la implementación. Por lo tanto, una lista se puede pensar como un arreglo dinámico.

Asumiendo que e es una expresión del mismo tipo que los componentes de la lista l, y que i es una expresión de tipo natural. Las operaciones con listas son las siguientes.

Operador	Descripción
e ▷ l	Agrega un elemento al comienzo de la lista
l ◁ e	Agrega un elemento al final de la lista
l . i	Devuelve el *i-ésimo* elemento de la lista
#l	Devuelve la longitud de la lista
head(l)	Devuelve el primer elemento de la lista
tail(l)	Devuelve la lista omitiendo el primer elemento
e ∈ l	Responde si un elemento pertenece a la lista
[ ]	Constante que simboliza la lista vacía

A continuación se implementan algunas funciones que utilizan las operaciones antes descriptas. Los elementos sintácticos introducidos en estos ejemplos, junto con sus significados semánticos serán explicados en detalle en sus correspondiente secciones.

```
fun sumatoria (L: list of nat) ret suma: nat
  suma := 0
  for l ∈ L do
    suma := suma + l
  od
end fun
```

La función *sumatoria* devuelve la sumatoria de todos los elementos de una lista. Itera sobre cada uno de los elementos de la misma, y almacena la suma acumulada en la variable *suma*.

```
fun inicializar_lista (n: nat) ret L: list of nat
  L := [ ]
  for i := n downto 1 do
    L := i ▷ L
  od
end fun
```

La función *inicializar_lista* construye una lista con los primeros n números naturales. Comenzando desde n, y disminuyendo hasta *1*, va agregando al comienzo de la lista cada uno de los elementos.

```
fun existe_par (L: list of nat) ret b: bool
  b := False
  for i := 1 to # L do
    b := b ∨ es_par(L.i)
  od
end fun
```

La función *existe_par* calcula si hay un número par en una lista determinada. La función *es_par* es una función auxiliar, que dado un natural afirma si el mismo es par o no.

4.2.3. Tipo Conjunto

Un tipo **set** es un estructura compuesta por una cantidad variable de componentes del mismo tipo. A diferencia de las listas, en un conjunto no existe un orden entre los elementos que lo conforman. La intuición detrás de los mismos, es igual a la de matemática.

Asumiento que P y Q son conjuntos del mismo tipo, y que e es una expresión cuyo valor es del mismo tipo que los componentes de los conjuntos. Las operaciones que se pueden realizar sobre conjuntos comprenden.

Operador	Descripción
P ∪ Q	Calcula la unión de conjuntos
P ∩ Q	Calcula la intersección de conjuntos
P − Q	Calcula la diferencia de conjuntos
| P |	Devuelve la cantidad de elementos del conjunto
e ∈ P	Responde si un elemento pertenece al conjunto
{ }	Constante que simboliza el conjunto vacío

A continuación, detallamos algunos ejemplos que emplean las operaciones anteriores. Al igual que con listas, las construcciones introducidas en los ejemplos serán explicadas en sus respectivas secciones.

```
fun dif_sim (A: set of T, B: set of T) ret C: set of T
  C := (A ∪ B) - (A ∩ B)
end fun
```

La función *dif_sim* calcula la diferencia simétrica de dos conjuntos. La misma es un conjunto compuesto por todos los elementos que solo pertenecen a uno de los dos operandos.

```
fun inicializar_conjunto (n: nat) ret C: set of nat
  C := {}
  for i := 1 to n do
    C := C ∪ {i}
  od
end fun
```

La función *inicializar_conjunto* construye un conjunto con los primeros n números naturales. Notar en esta función el uso del *azúcar sintáctico* para definir conjuntos. Simplemente se enumeran expresiones enceradas por llaves para denotar un conjunto.

4.2.4. Tipo Tupla

Un tipo **tuple** es una estructura compuesta por una cantidad fija de componentes, posiblemente de distintos tipos. Cada componente se denomina *campo*, y el identificador asociado a cada campo se le llama *alias*. Son utilizadas para empaquetar una serie de valores que se relacionan semánticamente. La única

operación disponible para trabajar con tuplas es la de obtener **.** un valor almacenado en la misma. Dada una tupla t y un alias a, se puede acceder al campo asociado a ese alias con $t.a$.

```
var t: tuple
       inicial: char,
       edad: nat,
       peso: real
       end tuple
```

En el ejemplo, se declara una variable t de tipo tupla. Los alias son *inicial*, que almacena un carácter, *edad*, que almacena un natural, y *peso*, que almacena un real. Notar el uso de las palabras claves **tuple** y **end tuple** para definir la misma.

```
t.inicial := 'm'
t.edad := 18
t.peso := 60.0
```

En el ejemplo anterior se almacenan distintos valores en los respectivos campos de la tupla previamente definida. Las tuplas tienen gran importancia para definir tipos de datos abstractos.

4.2.5. Tipo Puntero

Un tipo **pointer** es una estructura compuesta por un solo componente. Para el manejo dinámico de memoria se utilizan punteros. Con los mismos se reserva, aloja y libera memoria para el componente apuntado por el puntero, a lo largo de la ejecución del programa.

```
var p: pointer of nat
```

La declaración anterior crea una variable p que será un puntero. La misma servirá para direccionar un valor numérico natural en un futuro. Antes de realizar cualquier asignación sobre p, se deberá reservar memoria para el componente señalado, y una vez finalizado el cómputo se deberá liberar la memoria utilizada.

Asumiento que p es un puntero, y que a es un alias correspondiente a una posible tupla señalada por p. Las operaciones que manipulan punteros son las siguientes.

Operador	Descripción
alloc(p)	Reserva espacio en memoria para almacenar el componente
free(p)	Libera espacio en memoria apuntado por el puntero
$\star$p	Devuelve el componente señalado en memoria
p $\rightarrow$ a	Devuelve el campo asociado al alias de la tupla en memoria
null	Constante que simboliza un puntero que no apunta a nada

Como se puede observar, los punteros permiten manejar explícitamente direcciones de memoria y sus contenidos, por lo que permiten programar a bajo

nivel. Al igual que las tuplas, los punteros tienen gran importancia a la hora de definir tipos de datos abstractos. Como AlgoLang no posee *garbage collector* es responsabilidad del programador administrar el uso de memoria del programa.

```
alloc(p)
*p := 20
free(p)
```

En el ejemplo anterior, se utiliza *alloc* para reservar espacio en memoria para almacenar un valor natural y se coloca esa dirección en el puntero *p*. Luego, se almacena en esa misma dirección de memoria el valor *20*. Finalmente, se libera la memoria señalada por el puntero.

4.3. Definición de Tipos

Una característica muy importante de AlgoLang es la posibilidad de crear nuevos tipos de datos. El lenguaje hace uso del tipado fuerte, es decir, toda expresión posee un tipo definido. Debido a esto, se simplifica la interpretación del código y se fomenta la generación de código prolijo. Pero al momento de querer resolver problemas de mayor complejidad, es necesario poder extender los tipos del lenguaje con otro más sofisticados y robustos. Por lo tanto, AlgoLang ofrece ciertas construcciones con estos fines.

Para poder crear un nuevo tipo de datos se utiliza una sintaxis similar al siguiente ejemplo. En el mismo, se utiliza la palabra clave **type** para señalar la creación de un sinónimo de tipo. Le sucede el nuevo nombre de tipo, *matriz* en el ejemplo, y finalmente un tipo concreto del lenguaje.

```
type matriz = array[1..5, 1..5] of int
var M: matriz
```

Luego de la definición de tipo, se podrá utilizar el identificador *matriz* como un tipo concreto del lenguaje a lo largo del programa. El mismo, tendrá un comportamiento idéntico al arreglo al que se asoció. Por último, se puede observar que una vez definidos se pueden declarar variables de los tipos creados.

En el lenguaje también se permite definir lo que comúnmente se llama *tipos paramétricos*. Esto es una forma de definir nuevos tipos de datos, de la forma más general posible. Utilizando una variable de tipo (representada por un identificador), se puede crear un tipo nuevo donde su comportamiento no dependerá de la misma.

```
type stack of T = list of T
```

En el ejemplo, se creó un nuevo tipo de dato denominado *stack* (pila) que será una lista de tipo **T**. Con esta definición genérica podemos definir las operaciones del nuevo tipo pila, cualquiera sea el valor de la variable **T**.

```
proc empty (out p: stack of T)
  p := [ ]
end proc
```

```
proc push (in t: T, in/out p: stack of T)
  p := t ▷ p
end proc

fun top (p: stack of T) ret t: T
  t := head(p)
end fun

proc pop (in/out p: stack of T)
  p:= tail(p)
end proc

fun is_empty (p: stack of T) ret b: bool
  b := p = [ ]
end fun
```

Lo definido en el ejemplo, son todas las operaciones básicas de una pila. El proceso *empty* crea una pila vacía. El proceso *push* agrega un nuevo elemento al comienzo de la pila. La función *top* devuelve el primer elemento de la pila. El proceso *pop* elimina el primer elemento de la pila. Y la función *is_empty* pregunta si la pila está vacía.

Finalmente, hay que rescatar la posibilidad de definir tipos de datos recursivos. Los mismos, son aquellos donde el nombre del nuevo tipo figura también en la definición del mismo. Permitiendo definir así estructuras teóricamente infinitas. A continuación se muestra un ejemplo que utiliza las tuplas y punteros descriptos previamente.

```
type node = tuple
            value: nat,
            next: pointer of node
            end tuple
```

Para ilustrar, daremos un ejemplo donde se implementa una función que trabaja con el nuevo tipo de dato definido. Se puede pensar a la estructura *node* como una celda que almacena un valor natural y señala a otro *node*, conformando así una posible lista abstracta.

```
fun crear_lista_abs (n: nat) ret l : pointer of node
  var aux: pointer of node
  l := null
  for i := n downto 1 do
    alloc(aux)
    aux → value := i
    aux → next := l
    l := aux
  od
end fun
```

5. Expresiones

Las expresiones ocurren en asignaciones o en chequeos. Están compuestas por operadores y operandos. Producen un valor de cierto tipo, mediante la aplicación de los distintos operadores a sus respectivos operandos. No alteran el estado del programa, ya que no producen efectos secundarios.

5.1. Precedencia

Los distintos operadores para cada uno de los tipos básicos y estructurados ya se vieron en sus respectivas secciones. Cuando se emplean múltiples operadores en una expresión se utilizan las reglas de precedencia para determinar el orden de evaluación. En AlgoLang no hay ninguna especificación sobre la precedencia de operadores. A pesar de esto, se puede sugerir una tabla como la siguiente.

Operador	Precedencia	Categoría
$\neg$, $-$	Primera	Operadores Unarios
$*$, $/$, $\wedge$	Segunda	Operadores Multiplicativos
$+$, $-$, $\vee$	Tercera	Operadores Aditivos
$=$, $<$, $>$	Última	Operadores Comparativos

A la hora de evaluar se pueden utilizar las siguientes reglas.

- En operaciones con precedencias distintas, se evalúa primero la que tenga mayor precedencia.
- En operaciones con precedencias iguales, se evalúa primero la que se encuentra antes en el código.
- Si se utilizan paréntesis en expresiones, su contenido es evaluado primero.

5.2. Conversiones

A veces es necesario cambiar el tipo de una expresión, o una subexpresión, para poder hacer que la evaluación de la misma sea correcta. Es decir, queremos realizar una conversión para que los tipos sean compatibles.

```
var i: real
i := 1 + 2
```

Un ejemplo posible puede ser el anterior. En el mismo tenemos una suma de números naturales, pero la variable a asignar es de tipo real. En estas situaciones hay dos opciones. Informamos al programador sobre un posible error de tipos, o se realiza una conversión *implícita* del tipo de la expresión a un valor real. En el lenguaje no hay ninguna especificación sobre si las conversiones implícitas o explícitas son posibles. Esto comprende otra de las decisiones que se deberá tomar a la hora de la implementación del intérprete.

6. Declaración y Asignación de Variables

Como todo lenguaje imperativo, AlgoLang permite declaración de variables. Una variable representa un lugar en memoria explícitamente nombrado, que almacena un valor de cierto tipo. Para crear una variable nueva se utiliza la palabra clave **var**, seguida del identificador de variable y su tipo.

```
var i: nat
var j: bool
```

En el ejemplo anterior, se declaró una variable de tipo natural llamada *i*, y una variable de tipo bool llamada *j*.

En el lenguaje, las variable no son inicializadas luego de su declaración. Es responsabilidad del programador asignarles un valor inicial correspondientes a las mismas. Si en una hipotética ejecución del programa, se utiliza una variable sin inicialización entonces no se puede continuar con la ejecución del mismo. Para la asignación se utiliza el símbolo **:=**, precedido por el nombre de variable y sucedido por una expresión cuyo valor sea del mismo tipo que la variable a asignar.

```
i := 1
j := True
```

En la primer línea se asigna el valor *1* a la variable de tipo natural *i*, y en la segunda, el valor *True* a la variable booleana *j*.

La noción de estado es un concepto fundamental para la programación imperativa. Las variables son utilizadas para almacenar los resultados de la computación del programa. En el siguiente ejemplo, se evalúa la expresión de la derecha de la asignación y el resultado de la misma es almacenado en la variable booleana *j*.

```
j := j ∧ (i > 1)
```

Una característica de AlgoLang es la capacidad de declarar múltiples variables del mismo tipo en una línea. En el siguiente ejemplo, se declaran dos variables nuevas, *aux* y *tmp* de tipo real.

```
var aux, tmp: real
```

Hay distintas categorías de variables. Las variables se pueden dividir en cuatro clases en base a como se acceden a las mismas:

- **Variables Enteras** son representadas en su totalidad por un identificador. Es decir, poseen un único lugar en memoria.

  ```
  var pi: real
  pi := 3.14
  ```

- **Variables Indexadas** para obtener un componente de un arreglo se indexa de forma apropiada la variable que denota al mismo.

```
var A: array [1..5] of bool
A[1] := True
A[2] := False
```

- **Variables de Campo** para obtener un componente de una estructura se especifica el alias correspondiente al campo requerido.

```
var person: tuple inic char, edad: nat end tuple
person.inic := 'm'
person.edad := 18
```

- **Variables Referenciadas** para acceder al valor señalado por un puntero, se utiliza el operador de acceso con la variable que denota al puntero.

```
var p: pointer of char
alloc(p)
*p := 'g'
free(p)
```

7. Sentencias

La esencia de un algoritmo son las acciones que realiza. Estas acciones están contenidas en las sentencias del programa, las cuales se dicen ser ejecutables. Las mismas se dividen en sentencias simples, y sentencias estructuradas.

7.1. Sentencias Simples

Una sentencias simple es una sentencia cuyas partes no constituyen otras sentencias. Básicamente, son sentencias *atómicas* que forman la base para la construcción de las sentencias estructuradas.

7.1.1. Asignación

Las asignaciones le dan un valor a las variables, remplazando cualquier valor previo que pueden haber tenido. Se utiliza el operador **:=**, donde se evalúa la expresión a su derecha y se le asigna el valor a la variable de su izquierda.

```
var i: nat
i := 1
```

7.1.2. Llamada a Procedimientos

Las llamadas a subrutinas comprenden otra las sentencias simples. En la misma se denota el identificador correspondiente al procedimiento, y luego se listan entre paréntesis todos los parámetros actuales del mismo. Suponiendo que definimos un procedimiento para ordenar los valores de un arreglo *selection_sort*,

y tenemos un arreglo A de enteros, podemos realizar lo siguiente para invocar al procedimiento.

```
selection_sort(A)
```

7.2. Sentencias Estructuradas

Las sentencias estructuradas son construcciones compuestas por otras sentencias que pueden ser ejecutadas en secuencia, condicionalmente o repetidamente. Se pueden dividir en sentencias secuenciales, condicionales, e iterativas.

7.2.1. Sentencias Secuenciales

Muchos lenguajes de programación permiten la creación de *bloques de código*, compuestos por una serie de sentencias. Las mismas se deben de ejecutar de forma secuencial en el orden que aparecen en el código.

En AlgoLang no existe ninguna construcción explícita para denotar estas construcciones, pero si se pueden especificar secuencias de sentencias como en el siguiente ejemplo. En el mismo, se separan las distintas sentencias en líneas diferentes.

```
minp := min_pos_from(A, i)
swap(A, i, minp)
i := i + 1
```

El ejemplo anterior es un fragmento de un algoritmo de ordenación (*selection_sort*). En el mismo se realiza una asignación, una llamada a un proceso y finalmente otra asignación. Los identificadores *minp* e *i* son variables de tipo natural y A es un arreglo, mientras que *min_pos_from* y *swap* denotan una función y un procedimiento respectivamente.

7.2.2. Sentencias Condicionales

Todo lenguaje imperativo implementa algún tipo de expresión condicional. La misma sirve para realizar ejecuciones diferentes dependiendo del valor de una condición.

```
if i < n then
  minp := min_pos_from(A, i)
  swap(A, i, minp)
  i := i + 1
fi
```

En el ejemplo, si el valor de la variable i es menor al de n se ejecutará el bloque de código dentro del condicional. Caso contrario, las sentencias que se encuentran dentro del *if* no serán ejecutadas.

Para la sentencia condicional básica se usa la palabra clave **if** seguida de una expresión booleana (también denominada *guarda*), sucedida por la palabra

clave **then** y finalmente las instrucciones a ejecutar en el caso que la condición se cumpla. El bloque condicional finaliza con la palabra clave **fi**.

Además de la sentencia condicional compacta que vimos anteriormente, el lenguaje posee otros tipos más complejos de construcciones condicionales. Para el caso de querer realizar una división del flujo de ejecución más refinada se pueden especificar múltiples guardas utilizando **else if** e incluso se puede hacer uso de la palabra clave **else** para ejecutar un bloque de código cuando todas las demás guardas resultaron falsas.

```
var menor, igual, mayor: bool
menor := False
igual := False
mayor := False
if i < j then
  menor := True
else if i > j then
  mayor := True
else
  igual := True
fi
```

En el ejemplo se evalúa si la variable i es menor, mayor o igual a la variable j. Debido a que la ejecución del programa es secuencial se ejecutará el bloque de código cuya guarda sea la primera en ser satisfecha. Luego de esto, el programa saltará al final del condicional ignorando todas las otras guardas con sus respectivas instrucciones.

7.2.3. Sentencias Iteradoras

Las sentencias más importantes de un lenguaje imperativo son las que permiten ejecutar un conjunto de instrucciones una cantidad finita o infinita de veces. AlgoLang ofrece una amplia variedad de estas construcciones.

```
var A: array[1..5] of int
var i: nat
i := 1
while i ≤ 5 do
  A[i] := 1
  i := i + 1
od
```

El ejemplo inicializa los valores del arreglo en *1*. Esto lo hace, creando una variable i que toma distintos valores en el conjunto de índices del arreglo A e irá inicializando los valores del mismo. Notar que la sintaxis de esta instrucción consiste de la palabra clave **while** seguida de una expresión booleana (también denominada *guarda*), y luego un bloque de código encerrado entre las palabras claves **do** y **od**.

Otra sintaxis válida en el lenguaje, pero con el mismo significado semántico es la siguiente.

```
var A: array[1..5] of int
var i: nat
i := 1
do i ≤ 5 →
  A[i] := 1
  i := i + 1
od
```

La semántica intuitiva de esta construcción es la ejecución reiterada del bloque de código mientras la expresión booleana sea verdadera Esta abstracción nos permite realizar una serie de cambios de estados hasta obtener un resultado, momento donde la guarda deja de valer.

En otras situaciones, interesa ejecutar un conjunto de sentencias para distintos valores de una variable en un rango determinado. Es decir, si se quiere realizar de forma reiterativa ciertos cambios de estados conociendo de antemano la cantidad de veces que queremos iterar.

```
var A: array[1..5] of int
for i := 1 to 5 do
  A[i] := 1
od
```

Notar que con la sentencia **for**, uno puede especificar los límites de valores que puede tomar cierta variable dentro de su bloque. Hay una serie de consideraciones a tener en cuenta al utilizar esta instrucción.

- La variable a iterar es declarada de forma implícita en el mismo **for**, por lo que no es necesario especificar su tipo. En una futura implementación del lenguaje, se deberá tomar una decisión sobre si el tipo es inferido o debe ser especificado.
- El alcance (*scope*) estará restringido al bloque de la sentencia. Esto significa que una vez finalizada la ejecución de la instrucción la variable i dejará de ser accesible.
- En el cuerpo del **for** no se modificará el valor de la variable a iterar. La alteración del estado de i solo se realiza de forma implícita al finalizar cada iteración.

Hay más versiones de la sentencia **for**. A continuación mencionaremos algunas, y a lo largo del desarrollo de los temas siguientes señalaremos otras en la medida vaya siendo conveniente o necesario.

```
var A: array[1..5] of int
for i := 5 downto 1 do
  A[i] := 1
od
```

La semántica de este ejemplo es la misma a todos los ejemplos anteriores. La particularidad es que se puede especificar un rango descendente de valores para la variable de control. En los anteriores, i tomaba valores de forma ascendente.

Otra característica muy interesante, es que la instrucción **for** permite definir cualquier tipo enumerable para la variable iteradora. Esto permite que los límites de valores para la variable de control no sean estrictamente numéricos. Todo conjunto de valores que pueda ser enumerado puede ser utilizado para especificar los límites de la iteración.

```
var B: array['a'..'e'] of int
for k := 'a' to 'e' do
  B[k] := 1
od
```

Notar que para este ejemplo, se puede inferir el tipo de la variable k sin necesidad de especificar el mismo. En cambio para los ejemplos anteriores, con los límites *1* a *5* no queda claro si se refieren a los números naturales o a los enteros.

Finalmente, describiremos el uso de un iterador un poco más abstracto. Es muy común cuando trabajamos con arreglos acceder a los diferentes valores del mismo y operar luego con los mismos. Para tener un código con mayor nivel de abstracción, se puede usar algo similar al ejemplo siguiente.

```
var sumatoria: int
sumatoria := 0
for a ∈ A do
  sumatoria := sumatoria + a
od
```

Asumiendo que A es un arreglo de enteros. Este ejemplo suma todos los elementos del mismo. Dado que solo nos interesan los valores que almacena el arreglo, podemos calcular la sumatoria iterando directamente sobre el arreglo, ignorando de esta manera los índices.

8. Funciones

Para definir rutinas determinísticas, es decir, secuencias de instrucciones que no dependen del estado del programa, AlgoLang ofrece la posibilidad de especificar funciones. Las mismas realizan una computación, en base a un conjunto de parámetros, y devuelven un resultado. Las funciones son independientes del estado del programa, en el sentido que su comportamiento es determinado por los valores de entrada que recibe. Es importante notar que no modifican el estado de las variables que son pasadas como parámetros.

```
fun factorial (n: nat) ret fact: nat
  fact := 1
  for i := 1 to n do
    fact := fact * i
```

```
  od
end fun
```

En el ejemplo, se especifica una función que calcula el factorial de un número natural *n*. La variable *i* tomará distintos valores de *1* hasta *n*, mientras que en la variable *fact* se irá almacenando la productoria de estos números.

Para la sintaxis, se usa la palabra clave **fun** seguida del nombre de la función. Luego, entre paréntesis se especifican los parámetros de entrada separados por comas. Se tienen que detallar los tipos y los identificadores para cada una de las entradas que necesitará la función para su cómputo. También se suelen denominar *parámetros formales*. Con la palabra clave **ret** se especifica el valor de retorno, además del nombre y tipo de la variable asociada al mismo. Finalmente, en el cuerpo de la función se pueden usar todas las sentencias y expresiones que vimos hasta ahora para escribir el programa deseado. Para cerrar el bloque de la función, se utiliza **end fun**.

Sobre los identificadores de funciones y sus variables, no hay ninguna restricción especificada. Para la implementación del intérprete se deberá tomar una decisión sobre que nombres son válidos y en base a qué criterios se permitirán los mismos. Obviamente, no se podrán definir identificadores que también representen palabras claves.

```
fun es_par (n: nat) ret b: bool
  b := (n % 2) = 0
end fun

fun existe_par (A: array[1..n] of nat) ret b: bool
  b := False
  for a ∈ A do
    b := b ∨ es_par(a)
  od
end fun
```

En este último ejemplo se ilustran un par de características de las funciones. La función *es_par* calcula si un numero natural pasado como parámetro es par. En cambio, *existe_par* chequea si existe algún número par en el arreglo pasado como parámetro. Notar que para los arreglos, uno puede establecer límites fijos como en el caso del límite inferior *1*, o límites variables *n*. También se ve que para llamar una función anteriormente definida solo se especifica el nombre seguido entre paréntesis de todos sus parámetros.

Las funciones también pueden ser definidas de forma recursiva. Es decir, que en el mismo cuerpo de la función se puede llamar a sí misma para continuar con la computación.

```
fun es_par (n: nat) ret b: bool
  if n = 0 then
    b := True
  else if n = 1 then
    b := False
```

```
  else
    b := es_par(n - 2)
  fi
end fun
```

Este ejemplo muestra una manera no recomendable de definir la función *es_par* utilizando recursión. Se definen los dos casos bases, cuando *n* es *0* o *1* y si no se cumple ninguno se decrementa la variable en *2* realizando la llamada recursiva.

Finalmente, una última característica importante de las funciones es la definición polimórfica. En la misma, uno no especifica de manera concreta los tipos de las variables sino que utiliza una variable de tipo para poder crear una función cuya implementación pueda ser usada por más de un tipo.

```
fun indice_minimo (A: array[1..n] of T) ret min: nat
  min := 1
  for i := 1 to n do
    if A[i] < A[min] then
      min := i
    fi
  od
end fun
```

La función devuelve el índice en el arreglo del valor más chico. Se puede ver que se utiliza la variable de tipo **T** para especificar que la función puede tomar valores de cualquier tipo. En particular, esta función puede ser usada para encontrar el índice del mínimo de un arreglo de enteros, caracteres, naturales, entre otros (siempre y cuando la comparación esté definida para el conjunto de valores).

Una salvedad importante sobre las variables de tipos, es que no siempre pueden tomar *cualquier* tipo concreto. Usando el ejemplo anterior, la variable **T** no podría ser de tipo booleano. Esto se debe a que se utilizan operaciones de comparación entre valores de tipo **T**, y para los booleanos estas no están definidas. Esto comprenderá otra de las decisiones que se deberá tomar a la hora de la implementación del intérprete.

9. Procedimientos

Una construcción similar a las funciones son los procedimientos. Ambas representan formas de especificar rutinas reusables para realizar tareas particulares. Pero la diferencia fundamental entre ambos es que los procedimientos pueden modificar el estado del programa en su accionar. En base a un conjunto de parámetros (algunos de entrada, y otros de salida), un procedimiento realiza una computación que modificará el entorno del proceso que lo llamó.

```
proc p_sumatoria (in a, b: nat, out k: nat)
  var sum: nat
```

```
  sum := 0
  for i := a to b do
    sum := sum + i
  od
  k := sum
end proc

fun f_sumatoria (a, b: nat) ret k: nat
  k := 0
  for i := a to b do
    k := k + i
  od
end fun
```

Este ejemplo, muestra un programa donde se define una función *f_sumatoria* y un procedimiento *p_sumatoria* que hacen prácticamente lo mismo. Calcular la sumatoria de los números naturales desde a hasta b. La diferencia fundamental en estas dos construcciones es que la función devuelve un valor (el cual está almacenado en la variable local k), en cambio, el procedimiento está modificando el valor de una variable global k, que fue definida fuera del mismo.

La sintaxis de un procedimiento es similar al de una función salvo por el uso de las palabras claves **proc** y **end proc** que encierran al mismo, y la especificación de entradas y salidas. Para cada parámetro formal del procedimiento hay tres opciones:

- **in** determina que las variables de entrada serán utilizadas solo para lectura. Por lo tanto, su valor no será modificado.
- **out** significa que el valor asociado a la variable será alterado por la llamada al procedimiento. Su valor no será utilizado para la evaluación de una expresión.
- **in / out** establece que la variable va a ser empleada para lectura y escritura. Su valor inicial determinará la computación, pero el mismo será modificado a lo largo del procedimiento.

Al igual que en las funciones, los procedimientos permiten polimorfismo de la misma forma. En el siguiente ejemplo, se modifica un arreglo permutando los valores en las posiciones i y j. Notar que la variable de tipo **T**, se utiliza tanto en la especificación de parámetros como en la variable temporal dentro del procedimiento.

```
proc swap (in/out A: array[1..n] of T, in i, j: nat)
  var tmp: T
  tmp := A[i]
  A[i] := A[j]
  A[j] := tmp
end proc
```

Para contextualizar mejor el uso de procedimientos podemos observar el siguiente ejemplo. En el mismo se utilizará *swap* para invertir un arreglo. Se puede apreciar que luego de llamar al procedimiento auxiliar no se realiza ninguna asignación debido a que el valor del arreglo A es modificado de forma implícita por el mismo.

```
proc invertir_arreglo (in/out A: array[1..n] of T)
  for i := 1 to n / 2 do
    swap(A, i, n + 1 - i)
  od
end proc
```

Finalmente, y similar a funciones, se permite la definición de procedimientos recursivos. A continuación especificaremos el procedimiento análogo a la función *es_par* implementada anteriormente.

```
proc es_par (in n: nat, out b: bool)
  if n = 0 then
    b := True
  else if n = 1 then
    b := False
  else
    es_par(n, b)
  fi
end proc
```

10. Programas

Para finalizar, vamos a describir como se define un programa en AlgoLang. Como muchas de las características del lenguaje, no hay ninguna especificación precisa de la misma. Debido a esto, la siguiente descripción puede estar sujeta a modificaciones.

Para ejemplificar, vamos a implementar uno de los primeros algoritmos de ordenación vistos en la materia *selection_sort*. Algunas de las componentes del programa ya se ilustraron en secciones anteriores.

```
proc swap (in/out A: array[1..n] of T, in i, j: nat)
  var tmp: T
  tmp := A[i]
  A[i] := A[j]
  A[j] := tmp
end proc

fun min_pos (A: array[1..n] of T, i: nat) ret mp: nat
  mp := i
  for j := i + 1 to n do
    if A[j] < A[mp] then
```

```
      mp := j
    fi
  od
end fun

proc selection_sort (in/out A: array[1..n] of T)
  var minp: nat
  for i := 1 to n-1 do
    minp := min_pos(A, i)
    swap(A, i, minp)
  od
end proc
```

En el ejemplo se define un proceso *swap* que se encarga de intercambiar los valores de dos posiciones de un arreglo. La función *min_pos* calcula la posición del valor más chico entre un índice intermedio y el índice final del arreglo. El procedimiento *selection_sort* es el encargado de ordenar el arreglo llamando a los dos métodos auxiliares anteriores.

Un programa se describe en una serie de funciones y/o procedimientos que se especifican uno seguido del otro. No hay ninguna restricción sobre el orden en el que deben ser definidos. Ya que un programa de AlgoLang no puede ser ejecutado, no es necesario definir una función *main* como en C, o especificar sentencias fuera de cualquier bloque de código como en Python.