Introducción al AlgoLang

Matias Gobbi

13 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.

¡Buscar un nombre mejor para el lenguaje!

1. Introducción

El AlgoLang es un lenguaje imperativo 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 AlgoLang es poder introducir a los estudiantes a nuevos conceptos, y fomentar buenas técnicas de programación. Posee construcciones sintácticas de alto nivel, un formato estructurado y un sistema de tipos fuerte, lo que permite que los programas sean fáciles de leer e interpretar.

Hay que destacar que el lenguaje es un *pseudocódigo*. Por lo tanto, 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.

1.1. Características Principales

AlgoLang es un lenguaje teórico, esto significa que no existe ninguna implementación concreta del lenguaje, y tampoco hay una defición 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. Por lo tanto, AlgoLang permite trabajar con estructuras como los arreglos y manejar memoria de forma dinámica, pero también permite utilizar grafos y conjuntos para los algoritmos de mayor nivel. Obviamente, esto facilita la enseñanza de la materia pero posee una complicación fundamental para el desarrollo de un intérprete, la **ambigüedad**.

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 que funcionan para un rango de tipos distintos.

Debido a que no podemos ejecutar un programa de AlgoLang, ya que no existe ningún intérprete ni compilador para este lenguaje en pseudocódigo, 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 modulos, la división de código no resulta una necesidad para el desarrollo de la materia.

1.2. Tipos Nativos Simples

Comencemos propiamente con la descripción del lenguaje. El AlgoLang ofrece una variedad básica de tipos nativos. Cada uno de los mismos representa un conjunto de valores distinto y posee operaciones propias que los manipulan.

Los tipos numéricos son los siguientes: **nat** , **int** y **real** . Estos representan los conjuntos de los números naturales, números enteros y números reales, respectivamente. 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 $+\infty$. Los operadores que se ofrecen son los aritméticos y los de comparación clásicos.

Los aritméticos son los que evalúan a un valor numérico. Estos son la suma +, la multiplicación *, la resta -, la división /, el módulo %, y los operadores de máximo *max* y mínimo *min*.

```
@ 8 + 4
12
@ 8.0 * 4.0
32.0
@ 8 - 4
4
@ 8.0 / 4.0
2.0
@ 8 % 4
```

```
0
@ max (8, 4)
8
@ min (8.0, 4.0)
4.0
```

En cambio, los operadores de comparación son aquellos que evalúan a un valor booleano. Entre ellos están la igualdad $=$, el menor $<$, el mayor $>$, y sus respectivas negaciones.

```
@ 8 < 4
False
@ 8.0 > 4.0
True
@ 8 = 4
False
@ 8.0 ≤ 4.0
False
@ 8 ≥ 4
True
@ 8 ≠ 4
True
```

Notar que las operaciones para números son polimórficas. Esto significa que las funciones 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 el lenguaje no permite son las operaciones con distintos tipos de valores, es decir, no se puede sumar un natural con un entero, por ejemplo.

El tipo **bool** representa al conjunto de valores booleanos. Puede adoptar los valores *True* y *False*. También se ofrecen las operaciones clásicas para manejo de booleanos como la negación $\neg$, la disyunción $\vee$, y la conjunción $\wedge$.

En ninguna de las filminas utiliza operadores numéricos con argumentos de distinto tipo

```
@ ¬ True
False
@ True ∨ False
True
@ True ∧ False
False
```

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.

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.

```
@ 'a' = 'b'
False
@ '{' ≠ '}'
True
@ 'a' < 'z'
True
```

1.3. Declaración y Asignación de Variables

Como todo lenguaje imperativo, AlgoLang permite declaración de variables. Para crear una variable nueva se utiliza la palabra clave (keyword) **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*. 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 un valor del mismo tipo de 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*.

Finalmente, 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.

```
j := True ∧ (i > 1)
```

En este ejemplo, se evalúa la expresión de la derecha de la asignación y el resultado de la misma es almacenado en la variable booleana *j*. Se consulta el estado de la variable *i*, cuyo valor es *1*. Luego, al realizar la comparación obtenemos el valor *False*. Finalmente, al aplicar la conjunción obtenemos el valor *False* y actualizamos el estado de la variable *j* con este nuevo valor.

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 entero.

```
var aux, tmp: int
```

No vi ningún ejemplo de asignación múltiple. ¿Hay en el práctico?

1.4. Arreglos

Uno de los tipos más importantes de la materia es el **array**. Se puede pensar a un arreglo como una lista finita y estática de elementos, indexados por algún conjunto de valores enumerables. Para declarar una variable de tipo arreglo se necesitan el conjunto de los índices en los que estará definida y el tipo de los datos que almacenará.

```
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.

Un detalle importante a destacar es que el lenguaje AlgoLang es *case sensitive*. Esto significa que si una variable posee identificador a y otra identificador A entonces estamos tratando con dos variables distintas.

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. Hasta ahora solo vimos los números naturales, los enteros y los caracteres, pero más adelante veremos más ejemplos.

```
var B: array ['a'..'z'] of bool
var C: array [-4..12] of char
B['g'] := True
C[-2] := 'v'
```

En el ejemplo se declaran dos 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*.

Un par de observaciones que es importante rescatar en este ejemplo son: 1. Si el conjunto de índices de un arreglo es vacío, entonces el mismo tendrá longitud cero. Lo que sería similar a tener un tipo vacío. 2. En el lenguaje no está definido si se permiten mezclar declaraciones de variables y sentencias. A la hora de diseñar el intérprete se tendrá que tomar una decisión.

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

Nunca se usa D[1][1]. Al menos no en el teórico

El arreglo *D* posee dos dimensiones, ambas indexadas desde el número *1* al *5*. Para la asignación de valores del arreglo simplemente se separan con ',' las distintas coordenadas que se quieren acceder o actualizar. El arreglo *E*, 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*.

No dicen nada en el teórico sobre E[2, 'b'] :: array[10..20] of real, sería como una aplicación parcial de función, pero con arreglos :P

1.5. 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 < j then
  i := i + 1
fi
```

En el ejemplo, si el valor de la variable *i* es menor al de *j* se ejecutará la asignación aumentando en uno el valor de *i*. Caso contrario, la asignación que se encuentra dentro del *if* no es ejecutada.

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 expresiones 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
```

No utilizo ejemplos auto contenidos porque pensaba que se iban a extender demasiado algunos (y solo me quiero concentrar en presentar las instrucciones)

```
  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.

1.6. Iteradores

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 variabla 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.

1.7. 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.

No se usan nunca variables globales. Lo más lógico me parece es que no existan

```
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 **,**). Se tienen que detallar los tipos y los identificadores para cada una de las entradas que necesitará la función para su cómputo. 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.

```
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.

¡Feature Importante!

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.

1.8. Procedimientos

Una construccion 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 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 o sentencia.
- **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.

Lo que suelen hacer en el teórico, es declarar una variable sin inicializar y luego llamar a un proceso que la modifica

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
```

1.9. Tipos Nativos Complejos

Al comienzo de esta unidad se describieron los tipos nativos más simples, como los numéricos, los booleanos, o los arreglos. En esta sección explicaremos tipos más abstractos, los cuales no son fundamentales para el desarrollo de la materia, pero que son universalmente utilizados en los distintos lenguajes de programación existentes.

1.9.1. Listas

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.

Las operaciones que se pueden realizar sobre listas comprenden:

- Agregar $\triangleright$, que dado un elemento e de cierto tipo y una lista l del mismo tipo, agrega el elemento al comienzo de la lista.
- Agregar $\triangleleft$, que dada una lista l de cierto tipo y un elemento e del mismo tipo, agrega el mismo al final de la lista.
- Obtener **.**, que dada una lista l, y un natural i en el rango de índices válidos de la lista devuelve el *i-ésimo* elemento de la misma.
- Longitud $\#$, que dada una lista l devuelve la longitud de la misma.
- Cabeza *head*, que dada una lista l devuelve el primer elemento de la misma.
- Cola *tail*, que dada una lista l devuelve el resto de la lista omitiendo el primer elemento.
- Pertenece $\in$, que dada una lista l y un elemento e responde si el elemento pertenece a la misma (también existe su negación $\notin$).
- Lista Vacía [ ], es una constante que simboliza la lista vacía. Se pueden realizar comparaciones con la misma para chequear si una lista es vacía o no.

Un último detalle sobre listas. Al igual que en los arreglos, se encuentra definida una construcción abstracta del **for** para trabajar con los valores de una lista, ignorando sus índices. A continuación se implementan algunas funciones que utilizan las operaciones antes descriptas.

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

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

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 *sumatoria* devuelve la sumatoria de todos los elementos de una lista. La función *inicializar_lista* construye una lista con los primeros n números naturales. La función *existe_par* calcula si hay un número par en una lista determinada.

1.9.2. Tuplas

El siguiente tipo interesante son las tuplas, también denominadas estructuras. Son utilizadas para empaquetar una serie de valores de distintos tipos. La única operación disponible para trabajar con tuplas es la de obtener **.** un elemento. Dada una tupla t y un alias a, se puede acceder al valor 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*, *edad*, y *peso*, de tipos **char**, **nat**, y **real**, respectivamente. Notar el uso de las palabras claves **tuple** y **end tuple** para definir la misma. Las tuplas tendrán mayor importancia cuando se detalle como definir nuevos tipos de datos.

1.9.3. Conjuntos

Uno de los tipos de más alto nivel que ofrece AlgoLang, es el de conjuntos. La intuición detrás de los mismos, es igual a la de matemática. Las operaciones que se pueden realizar sobre conjuntos comprenden:

- Unión $\cup$, que dados dos conjuntos *c1* y *c2*, aplica la unión de conjuntos.
- Intersección $\cap$, que dados dos conjuntos *c1* y *c2*, aplica la intersección de conjuntos.
- Diferencia -, que dados dos conjuntos *c1* y *c2*, aplica la diferencia de conjuntos.
- Cardinal $|\ |$, que dado un conjunto *c*, devuelve la cantidad de elementos que posee.
- Pertenece $\in$, que dado un conjunto *c* y un elemento *e* responde si el elemento pertenece al mismo (también existe su negación $\notin$).
- Conjunto Vacío { }, es una constante que simboliza el conjunto vacío. Se pueden realizar comparaciones con el mismo para chequear si un conjunto es vacío o no.

A continuación, detallamos algunos ejemplos que emplean las operaciones anteriores.

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

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 *dif_sim* calcula la diferencia simétrica de dos conjuntos. La función *inicializar_conjunto* construye un conjunto con los primeros *n* números naturales.

1.9.4. Punteros

Finalmente, uno de los tipos complejos más usados en la materia. Para el manejo dinámico de memoria se utilizan los punteros para reservar, alojar y liberar memoria para distintas estructuras de datos a lo largo de la ejecución de un 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.

Las operaciones que manipulan punteros son las siguientes:

- Alojar *alloc*, que dado un puntero p, reservará espacio en memoria para almacenar la estructura de datos que será apuntada por p y asignará la dirección en el puntero.
- Liberar *free*, que dado un puntero p, liberará el espacio de memoria apuntado por el puntero.
- Puntero Vacío *null*, es una constante utilizada para señalar que un puntero no apunta a ninguna dirección válida de memoria.
- Acceder $\star$, que dado un puntero p, devuelve el valor en memoria señalado por el mismo.
- Acceder Tupla $\rightarrow$, que dado un puntero p que apunta a una tupla en memoria, y un alias a, devuelve el valor asociado a ese alias en la tupla en memoria señalada por p.

Como se puede observar, los punteros permiten manejar explícitamente direcciones de memoria y sus contenidos, por lo que permiten programar a bajo nivel. Como AlgoLang no posee *garbage collector* es responsabilidad del programador administrar el uso de memoria del programa.

1.10. Creación de Nuevos Tipos

Para finalizar la introducción al AlgoLang es necesario hablar de la creación de 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
```

Luego de la definición de tipo, se podrá utilizar el nombre *matriz* como un tipo concreto del lenguaje a lo largo del programa. El mismo, tendrá un comportamiento idéntico al arreglo al que se asoció.

Otra forma de declarar nuevos tipos es mediante los tipos enumerados. En el mismo, se enumeran todos los valores posibles (entre paréntesis) que puede adoptar este nuevo tipo de dato.

```
type dia = (dom, lun, mar, mie, jue, vie, sab)
type semana = array [dom..sab] of real
var d: dia
var S: semana
```

Al utilizar paréntesis, todos los nombres que se coloquen separados por **,** formarán parte de los nuevos valores del tipo. Notar que los tipos enumerados tienen la capacidad de poder ser utilizados como índices de arreglos. 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, 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 operacionas básicas de una pila. El proceso *empty* crea una pila vacía. El proceso *push* agrega un nuevo elemento a 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 anteriormente.

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

Para finalizar, 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
```