

Introducción al FritoLang

Matias Gobbi

September 22, 2019

Fundamentos

Antes de comenzar propiamente con la descripción y desarrollo de nuestro intérprete, lo correcto sería dar una introducción al funcionamiento del lenguaje. Por lo tanto, en la siguiente sección nos dedicaremos a dar una explicación intuitiva de las distintas construcciones que ofrece el FritoLang.

1 Introducción

El FritoLang es un lenguaje imperativo teórico 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 del análisis de algoritmos, la creación de estructuras de datos y la comprensión de algoritmos avanzados.

Con el FritoLang se busca poder introducir a los estudiantes a nuevos conceptos, y fomentar buenas técnicas de programación. El lenguaje posee construcciones sintácticas sencillas, un formato estructurado y un sistema de tipos fuerte, lo que permite que los programas sean fáciles de leer e interpretar.

1.1 Características Principales

Como mencionamos al comienzo, el FritoLang es un lenguaje teórico. Esto significa que no existe ninguna implementación concreta del lenguaje, lo cual posee ciertas consecuencias importantes. La más inmediata es que el lenguaje es flexible en cuanto al nivel de abstracción que maneja. A lo largo de la materia se emplea el FritoLang variando el mismo. Se diseñan algoritmos que trabajan con arreglos y manejo dinámico de memoria, pero también se permite utilizar grafos y conjuntos para los conceptos más complejos. 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 *real*. Al mismo

tiempo, el lenguaje ofrece varias construcciones polimórficas que funcionan para un rango de tipos distintos.

Al ser un lenguaje formativo, el FritoLang 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.

1.2 Tipos Nativos Simples

Ahora comenzaremos propiamente con la descripción del lenguaje. El FritoLang ofrece una gran variedad 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. 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, 4)
4
```

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

```

Hay que 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 .

```

> ¬ True
False
> True ∨ False
True
> True ∧ False
False

```

Un detalle importante que hay que notar en estas operaciones es el uso de la llamada evaluación por *short-circuit*. 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, 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, FritoLang permite declaración de variables. Para crear una variable nueva se utiliza la keyword **var**, seguida del identificador

de variable y el tipo de datos que almacenará la misma.

```
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 correspondientes a las mismas. 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
```

Siguiendo con el ejemplo, se asigna el valor 1 a la variable de tipo natural i , y 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  $\wedge$  ( i > 1 )
```

En el ejemplo anterior, se evalúa la expresión de la derecha de la asignación y el resultado de la misma es almacenado en la variable booleana j . Al remplazar el valor almacenado por la variable, el estado del programa fue modificado. Esta construcción posee un rol fundamental en el desarrollo de programas imperativos.

Cuando se tienen que declarar múltiples variables del mismo tipo, se puede utilizar declaración múltiple. En el ejemplo siguiente, se declaran dos variables nuevas, aux y tmp de tipo entero.

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

1.4 Arreglos

Uno de los tipos más importantes de la materia es el **array**. Se puede pensar a un arreglo como una lista estática de elementos indexados. Para declarar un arreglo se necesita el rango de los índices en los que el mismo estará definido 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
```

Lo anterior crea un arreglo A que estará definido para los índices $1, 2, 3$, y 4 . Luego, en cada posición i del arreglo se almacenará la sumatoria de los primeros i números naturales.

Otra forma de hacer lo mismo, haciendo uso de la noción de estado es la siguiente:

```

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

```

En el ejemplo se declaran dos arreglos distintos. Los índices del arreglo *B* son todos los caracteres ordenados alfabéticamente desde la letra *a* hasta la *z*. Notar que este es un arreglo de booleanos. El otro arreglo *C* tendrá índices que irán desde el número -4 hasta el 12. Este último es un arreglo de caracteres.

Por último, hay que mencionar que los arreglos en FritoLang 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
D[1, 1] := 4
var E: array [1..3, 'a'..'e', 10..20] of real
E[2, 'b', 15] := 4.0

```

Se puede ver que 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. 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.

1.5 Condicionales

Otra de las construcciones básicas que todo lenguaje imperativo implementa es el condicional. El mismo sirve para separar el flujo de ejecución del programa en base a si una condición es satisfecha o no.

```

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, se finalizará la ejecución del programa.

Obviamente, FritoLang ofrece otras varias construcciones para utilizar condicionales además de la ilustrada anteriormente. Para el condicional básico se usa la keyword **if** seguida de una expresión booleana, sucedida por la palabra clave **then** y finalmente las instrucciones a ejecutar en el caso que la condición se cumpla. 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 keyword **else** para ejecutar un bloque de código cuando todas las otras guardas no se cumplieron.

```
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.

1.6 Iteradores

Las intrucciones más importantes de un lenguaje imperativo son las que permiten realizar una serie de comandos de forma iterativa. FritoLang 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
```

En el ejemplo, creamos una variable i que tomará distintos valores en el rango del arreglo A e irá inicializando los valores del mismo en 1 . Notar que la sintaxis de esta instrucción consiste de la keyword **while** seguida de una expresión booleana, y luego un bloque de código encerrado entre las palabras claves **do** y **od**.

Otra notación diferente a la anterior pero con el mismo significado es la siguiente. También válida en el lenguaje.

```

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 las dos construcciones anteriores, es que se ejecutará el segmento de código de forma reiterada, siempre y cuando la expresión booleana siga siendo verdadera. Esta abstracción nos permite realizar una serie de cambios de estados de forma extensiva, mientras que la condición se satisfaga.

En otras situaciones, interesa solamente ejecutar una serie de instrucciones 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 los mismos.

```

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

```

Notar que con la keyword **for**, uno puede especificar los límites de valores que puede tomar cierta variable dentro del segmento de código siguiente. Hay una serie de consideraciones que hay que tener 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 ya que el mismo es inferido de los límites especificados en la instrucción.
- La visibilidad de la variable estará restringida a la ejecución del ciclo. 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 de control. La alteración del estado de *i* solo se realizará al iniciar cada bucle y se limitará al rango especificado.

Hay muchas más formas de especificar la instrucción anterior. A continuación mencionaremos algunas, y a lo largo del desarrollo de los temas siguientes señalaremos otras en la medida que sea adecuado.

```

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.

Otra característica muy interesante es que la instrucción **for** es polimórfica. 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.

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 más prolijo, 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. De esta forma, como solo nos interesan los valores que se almacenan en el arreglo, podemos calcular la sumatoria de todos sus elementos ignorando los índices.

1.7 Funciones

Para definir rutinas determinísticas a fin de automatizar una tarea particular, FritoLang 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. También es importante notar que no modifican el estado de la computación.

```
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 . Para la sintaxis, se usa la keyword **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 keyword **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 instrucciones que vimos hasta ahora para calcular el resultado deseado.

```
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, dado un número natural, si el mismo es par. En cambio, *existe_par* chequea si dado un arreglo de naturales, alguno es par. 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
```

Una forma (muy ineficiente) de definir la función *es_par* es como en el ejemplo de arriba. 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
    end if
  end for
end fun
```

```

    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.

1.8 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)
  k := 0
  for i := a to b do
    k := k + i
  od
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

```

En lo anterior, se muestra un ejemplo trivial 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 keywords **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 realización del cómputo.

- **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
```

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 idéntica a un arreglo, salvo por el hecho que su longitud no está predefinida. Por lo tanto, una lista se puede pensar como un arreglo dinámico cuyo tamaño puede ir variando a lo largo de la ejecución de un programa. 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 \cdot , 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 descritas.

```
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

```

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. Las tuplas tendrán mayor importancia cuando se detalle como definir nuevos tipos de datos.

1.9.3 Conjuntos

Uno de los tipos más abstractos que ofrece FritoLang, 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.

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 to 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 el puntero y asignará la dirección en p .
- 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 FritoLang 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 FritoLang 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 otros más sofisticados y robustos. Por lo tanto, FritoLang 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 keyword **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 que puede adoptar este nuevo tipo de dato.

```
type semana = (dom, lun , mar, mie , jue , vie , sab)
type calendario = array [dom..sab] of real
val A: calendario
val 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.

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 to node
    end tuple
```

Para finalizar, daremos un ejemplo donde se implementa una función que trabaja con el nuevo tipo de dato definido anteriormente. 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 to node
  var aux: pointer to node
  l := null
  for i := n downto 1 do
    alloc(aux)
    aux → value := i
    aux → next := l
    l := aux
  od
end fun
```