

toodots

Entrega final: Práctica de Procesadores de Lenguajes

Martín Fernández de Diego
Belén Sánchez Centeno

1 Introducción

Tras analizar varios lenguajes de programación poco comunes, hemos visto uno que nos ha llamado especialmente la atención: **Chef**.

Chef es un lenguaje de programación extremadamente esquemático en cuanto a su estructura pero se expresa con una retórica en demasiado alto nivel.

Vamos a intentar desarrollar un lenguaje inspirado en **Chef** pero atajando el uso excesivo de lenguaje natural con una simbología más abreviada –con muchos puntos– y respetando ciertas reglas intuitivas que iremos describiendo a lo largo del documento.

2 Especificación del lenguaje

2.1 Identificadores y ámbitos de definición

El bloque primitivo, el fichero que alberga todo el código del programa, poseerá dos zonas claramente diferenciadas: una zona inicial a modo de *Ingredientes* que denotaremos con el identificador **OBJECTS**, donde se declararán las variables globales y funciones, y otra zona similar a la *Preparación* dada por el identificador **PROCESS**, análoga a la función *main* de C, pero en este caso, con mucho más protagonismo. La ejecución terminará al encontrar el EOF. Con esta estructura, establecemos simplicidad en el código y disminuimos las restricciones por orden de aparición.

El fin de cada instrucción en nuestro lenguaje se indicará con una coma **,**.

Comentarios

Los comentarios podrán constar de un número indefinido de líneas porque siempre estarán acotados por los símbolos de apertura **((** y de cierre **))**. No estará permitida la anidación de comentarios.

```
((Esto es un comentario de prueba))
```

Variables simples

La creación de variables simples será de la forma **- nombreVariable:tipo**. Podrán declararse variables tanto en la zona **OBJECTS**, las globales, como en la **PROCESS**, las locales.

```
- x:Int, ((Declaramos la variable x de tipo entero))
```

Variables complejas

La declaración de arrays se efectuará colocando el tipo entre corchetes y estableciendo su tamaño a continuación entre paréntesis.

Serán de la forma `- nombreVariable:[tipo] (tamaño)`.

El acceso a una posición del array se hará de la forma `nombreVariable[pos]`.

```
- v:[Char] (10), ((Declaramos la variable v de tipo array de 10 elementos))
```

Punteros

Los punteros se definirán, además de con `-` como las variables ordinarias, con el símbolo `>` de la siguiente forma `- >nombrePuntero:tipo`. Por tanto, accederemos al contenido de la variable con el comando `>nombrePuntero` y a su dirección con simplemente `nombrePuntero`.

Se accederá a la dirección de memoria de una variable ordinaria de la siguiente forma `<nombreVariable`.

```
((Declaramos una variable a y un puntero b del mismo tipo))
- a:Char := "a",
- >b:Char,

b := <a, ((Apuntamos con el puntero b a la direccion de a))
>b := "b", ((Cambiamos el contenido de la variable a de 'a' a 'b'))
```

Bloques anidados

Los bloques comenzarán con `:` y terminarán con el símbolo `.`.

```
+ fun:Int(v:[Int],i:Int,j:Int):
- elem:Int,
  elem := v[i]+v[j],
} elem,
.
```

Funciones

Como ya hemos dicho, las funciones deberán ser declaradas en la zona `OBJECTS`, a continuación de las variables, indicando su nombre, el tipo de la variable que devuelven y sus argumentos junto a sus tipos. Tendrá la forma `+ nombreFuncion:tipoFuncion(arg1:tipoArg1,...,argn:tipoArgn)`. Es importante indicar con el símbolo `+` la declaración de funciones para poder distinguirlo así de la declaración de variables. No estará permitido que haya más de una función con el mismo nombre aunque cambien el tipo o el número de argumentos.

Devolveremos el valor del tipo de la función mediante el símbolo reservado `}` de la forma `}` `valor`.

```

((Declaramos una funcion))
+ fun:Char(sel:Int,arg1:Char,arg2:Char):
    - var:Char,
    ? (sel == 1): var := arg1, .
    ! : var := arg2, .
    } var, ((Devolvemos la solucion de la funcion))
.

```

Clases

Se considerará *clase* al fichero independiente cuyo cuerpo solo contenga la zona inicial de **OBJECTS**. Será suficiente esa zona, donde describiremos los atributos de la clase con variables globales y los métodos con funciones.

La importación de clases al programa se efectuará en las primeras líneas previas a la zona **OBJECTS** mediante el símbolo **#** de la forma **# nombreClase**.

Para instanciar una clase será suficiente con declarar una variable del tipo de la clase y asignarle una llamada a su constructora.

```

# Stack.class

OBJECTS
- s:Stack := Stack(10),

PROCESS
((Vaciamos la pila))
@ (s.size > 0):
    s.pop(),.
.

```

2.2 Tipos

Utilizaremos los tipos usuales de acuerdo a los lenguajes de programación más comunes y las operaciones típicas.

Tipos básicos

- Enteros: Identificados mediante la palabra reservada **Int**.
- Reales: Identificados por la palabra reservada **Float**. Separaremos la parte entera de la decimal con el símbolo **.**.
- Booleanos: Identificados por **Bool** y con los dos valores predefinidos **true** y **false**.
- Caracteres: Identificados por la palabra reservada **Char**. Denotamos a los caracteres entre comillas dobles de la forma **"a"**. También estableceremos que los arrays de caracteres **[Char]** se escriban de la forma **"abc"** en lugar de **["a","b","c"]**.
- Sin tipo: Identificados por **Void**. Utilizado para las funciones que no devuelven valor.

Tipos complejos

- Enumerados: Se identifican con la palabra reservada `Enum` y se muestran como un bloque anidado con un conjunto de identificadores - `nombreEnumerado:Enum: obj1,...,objn..`. Para declarar una variable del tipo enumerado que hemos definido se utiliza el nombre del enumerado en el tipo - `nombreVariable:nombreEnumerado`. Para dar valor con el objeto i a dicha variable le asignamos uno de los identificadores `nombreVariable := obji`.

```
((Declaramos Colors como un Enum con tres valores))
- Colors:Enum: RED,BLUE,GREEN.
((Declaramos la variable r del tipo Colors con un valor inicial))
- r:Colors := RED,
```

- Estructurados: Se identifica mediante la palabra `Struct` y se describe de la siguiente forma - `nombreEstructurado:Struct: - atri1:tipo1,...,- atrin:tipon,..`. Para declarar una variable del tipo enumerado que hemos definido se utiliza el nombre del enumerado en el tipo - `nombreVariable:nombreEstructurado`. Cuando queramos acceder al atributo i , escribimos `nombreVariable.atri`.

```
((Declaramos Data como un Struct con atributos id y value))
- Data:Struct:
  - id:[Char],
  - value:Int,
.,
((Declaramos la variable person del tipo Data))
- person:Data,
((Asignamos a person el id "Eva"))
- person.id := "Eva",
```

Operadores

- Operadores lógicos: Definimos el unario `not` y los binarios `and` y `or`.
- Operadores aritméticos: Definimos los operadores binarios `+`, `-`, `*`, `/` y `%`.
- Operadores relacionales: Definimos los operadores binarios `<`, `>`, `<=`, `>=`, `==` y `><`.

2.3 Conjuntos de instrucciones del lenguaje

Asignación

En `toodots` la asignación queda determinada por su símbolo matemático estricto `:=`. De esta forma, evitamos confusiones con la equivalencia lógica. La asignación, por supuesto, es válida tanto para variables simples como complejas y puede realizarse durante la propia declaración de las mismas. Sin embargo, producirá un error si la variable no está declarada o si los tipos no coinciden.

Prioridad	Operación	Significado	Asociatividad
1	not	Negación	derecha
2	* / %	Multiplicación, división y módulo	izquierda
3	+ -	Suma y resta	izquierda
4	< > <= >=	Menor, mayor, menor o igual y mayor o igual	izquierda
5	== ><	Igual y distinto	izquierda
6	and	Conjunción lógica	izquierda
7	or	Disyunción lógica	izquierda

Tabla de prioridades de los operadores del lenguaje

```

- x:Bool := true,
x := false,
- w:[Int] := [9, 1, 2, 3],
w[0] := 0,

```

Condicional de una rama

La estructura básica de una función condicional de una rama vendrá dada por el pseudocódigo `? (condicion): bloque..`

```

((Declaramos una funcion que devuelve 1 si le llega true y 0 en cc))
+ selectBool:Int(b:Bool):
  - count:Int := 0,
  ? (b == true): count := 1,.
  } count,
.

```

Condicional de dos ramas

Análogamente, la función condicional de dos ramas estará descrita por la siguiente estructura `? (condicion): bloque. ! : bloque..`

De esta forma, el `?` de toodots corresponde con el `if` de C y el `!` con el `else`.

```

((Declaramos una funcion que devuelve 1 si x>0 y 0 si x=0))
+ selectInt:Int(x:Int):
  - count:Int,
  ? (x > 0) : count := 1,.
  ! : count := 0,.
  } count,
.

```

Bucle

De momento, estará definido únicamente un tipo de bucle, el `@`. Un bucle sencillo que nos aportará todo el potencial. Su estructura quedará definida por `@ (condicion): bloque.`

```
((Declaramos una funcion que cuenta un numero de horas indicadas))
+ sleep:Void(hours:Int):
    - count:Int := 0,
    @ (count < hours):
        count := count + 1,
    .
.
```

Llamadas a funciones

La llamada a las funciones se ejecutará escribiendo simplemente el nombre de la función de la forma `nombreFuncion(arg1, ..., argn)`. Su resultado podrá ser acumulado en una variable con una asignación usual o bien utilizado directamente dentro de expresiones. Gracias a la estructuración de nuestro código, no importa el orden de definición de las funciones ni el momento en el que son llamadas porque el compilador ha leído de antemano todos los *ingredientes* de la zona `OBJECTS`.

Llamadas a métodos de clases

La llamada a los métodos de una clase arbitraria se ejecutará declarando primero una variable de la clase `nombreVariable:nombreClase` y después `nombreVariable.nombreFuncion(arg1, ..., argn)`.

Imprimir

La ejecución de la instrucción `>>` seguida de una expresión de tipo válido, mostrará por pantalla el resultado.

```
((Hola Mundo))
OBJECTS
- hola:[Char] := ["H", "o", "l", "a", " ", "M", "u", "n", "d", "o", "#"],
PROCESS
- i:Int := 0
@ (hola[i] >< "#") :
    >> holaMundo[i],
    i = i + 1,
.
```

3 Ejemplos

Serie de Fibonacci

OBJECTS

```
- number: Int := 10, ((Longitud de la solucion deseada))
- sol:[Int](number), ((Variable donde se almacena la solucion))
- n1: Int := 0,
- n2: Int := 1,
- n3: Int,
- i: Int := 2,
```

PROCESS

```
sol[0] := n1,
sol[1] := n2,
@ (i < number):
    n3 := n1+n2,
    sol[i] := n3,
    n1 := n2,
    n2 := n3,
    i := i+1,
```

.

Ordenamiento de la burbuja

OBJECTS

```
- a:[Int] := [64,34,25,12,22,11,90], ((Array desordenado))
- sol:[Int](7), ((Variable donde se almacena la solucion))
+ bubbleSort:[Int]():
```

```
    - i: Int := 0,
    - j: Int := 0,
    @ (i < l-1):
        @ (j < l-i-1):
            ? (a[j] > a[j+1]): swap(a,j,j+1) .
            j := j+1,
        .
        i := i+1,
    .
    } a,
```

```
    .
+ swap: Void(a:[Int],x: Int,y: Int):
    - temp: Int := a[x],
    a[x] := a[y],
    a[y] := temp,
```

.

PROCESS

```
sol := bubbleSort(),
```

Clase *stack* de tipo entero

((Ejemplo de definicion de la clase pila))

OBJECTS

```
- MAX: Int := 1000,
- array: [Int](MAX),
- size: Int := 0,
- it: Int := -1,
+ push: Void(elem: Int):
    ? (size < MAX):
        it := it+1,
        array[it] := elem,
        size := size+1,
    .
+ pop: Void():
    ? (size > 0):
        it := it-1,
        size := size-1,
    .
.
+ top: Int():
    - elem := -1,
    ? (size > 0):
        elem := array[it],
    .
    } elem,
.
.
```
