`toodots`
Entrega 1: 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; es decir, 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, funciones, clases, etc.; 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 `END` o volverá a ejecutarse al encontrar `AGAIN`. Con esta estructura, establecemos simplicidad en el código y disminuimos las restricciones por orden de aparición.

Los bloques anidados también harán la declaración de variables locales en la parte superior de su definición. Aunque esto sea solo un aspecto puramente convencional.

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`.

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

Variables complejas

La declaración de arrays se efectuará colocando el tipo entre corchetes y estableciendo su tamaño a continuación entre paréntesis.
Los de una dimensión serán de la forma `- nombreVariable:[tipo](tamaño)`.
Los de varias dimensiones se definirán intuitivamente mediante la anidación de n corchetes de la forma `- nombreVariable:[...[tipo]...](tamaño`$_1$`)...(tamaño`$_n$`)`.
El acceso a una posición del array se hará con tantos corchetes seguidos como dimensiones tenga `nombreVariable[pos`$_1$`]...[pos`$_n$`]`.

```
((Declaramos la variable v de tipo array unidimensional de 10 elementos))
- v:[Char](10),
((Declaramos la variable w de tipo array bidimensional de 3x3 elementos))
- w:[[Bool]](3)(3),
```

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:Void(v:[Int],i:Int,j:Int):
    ((OBJECTS))
    - elem:Int,

    ((PROCESS))
    elem := v[i] + v[j],
    } elem,
.
```

Funciones

Como ya hemos dicho, las funciones deberán ser declaradas en la zona superior del bloque primitivo, a continuación de las variables, indicando su nombre, el tipo de la variable que devuelven y sus argumentos junto a sus tipos. Desarrollaremos la función en el fichero de la forma `+ nombreFuncion:tipoFuncion(`$\texttt{arg}_1$`:tipoArg`$_1$`,...,`$\texttt{arg}_n$`:tipoArg`$_n$`)`. 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,arg3:Char):
    ((OBJECTS))
    - var:Char,

    ((PROCESS))
    ?! (sel):
        ? (1): var := arg1, .
        ? (2): var := arg2, .
        ! : var := arg3, .
    .
    } 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`.

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: Indentificados 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: Indentificados 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: obj`$_1$`,...,obj`$_n$`.`.
  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 := obj`$_i$.

```
((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: - atri`$_1$`:tipo`$_1$`,...,- atri`$_n$`:tipo`$_n$`,.`.
  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`$_i$.

```
((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 unarios `++` y `--` y también los operadores binarios `+`, `-`, `*`, `/` y `mod`.
- Operadores relacionales: Definimos los operadores binarios `<`, `>`, `<=`, `>=`, `==` y `><`.

Prioridad	Operación	Significado	Asociatividad
0	`++ --`	Incremento y decremento	derecha
1	`not`	Negación	derecha
2	`* / mod`	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 or`	Conjunción y disyunción lógicas	izquierda

Tabla de prioridades de los operadores del lenguaje

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.

```
- x:Bool := true,
x := false,
- w:[[Int]] := [[0,1],[1,2],[2,3]],
w[0][0] := 4,
```

Condicional con una y dos ramas

La estructura básica de una función condicional de una o dos ramas vendrá determinada por el pseudocódigo `? (condicion): bloque anidado. ! : bloque anidado.`.

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

Condicional de salto

La función típica del switch aprovecha el léxico de la condicional para quedar determinada como `?! (variable): ? (valor): bloque. ... ? (valor): bloque. ! : bloque.`.

```
((Declaramos un enumerado con cuatro direcciones))
- Directions:Enum: UP, DOWN, LEFT, RIGHT.,

((Declaramos una funcion que asigna puntuaciones a las direcciones))
+ directionToInt:Int(dir:Directions):
    - count:Int := 0,
    ?! (dir):
        ? (UP): count := 2 .
        ? (RIGHT): count := 1 .
        ? (LEFT): count := 1 .
        ! : count := -1 .
    .
    } count,
.
```

Bucle

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

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

Llamadas a funciones

La llamada a las funciones se ejecutará escribiendo simplemente el nombre de la función de la forma `nombreFuncion(arg`$_1$`,...,arg`$_n$`)`. Su resultado podrá ser acumulado en una variable con una asignación usual.
Gracias a la estructuración de nuestro código, no importa el orden de la 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.nombreFunción(arg`$_1$`,...,arg`$_n$`)`.

2.4 Gestión de errores

Se establecerán técnicas de detección y resolución de errores indicando exactamente el lugar donde se producen. Ampliaremos esta descripción durante el desarrollo del lenguaje.

3 Ejemplos

Hola mundo

```
# InOut,

OBJECTS
- io:InOut,

PROCESS
io.print("Hello world!"),

END
```

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,
loop(i < number):
    n3 := n1+n2,
    sol[i] := n3,
    n1 := n2,
    n2 := n3,
    ++i,
.

END
```

Ordenamiento de la burbuja

```
OBJECTS
- l:Int := 7 ((Longitud del array))
- aInicial:[Int](l) := [64,34,25,12,22,11,90], ((Array desordenado))
- sol:[Int](l), ((Variable donde se almacena la solucion))
+ bubbleSort:[Int](a:[Int]):
    - i:Int := 0,
    - j:Int := 0,
    loop (i < l-1):
        loop (j < l-i-1):
            ? (a[j] > a[j+1]): swap(a,j,j+1) .
            ++j,
        .
        ++i,
    .
    } a,
.
+ swap:Void(a:[Int],x:Int,y:Int):
    - temp:Int := a[x],
    a[x] := a[y],
    a[y] := temp,
.

PROCESS
sol := bubbleSort(aInicial),

END
```