Especificación del lenguaje BARTOLO

Alejandro Ramírez Rodríguez
David Seijas Pérez

27 de mayo de 2021

Índice

1 Identificadores y ámbitos de definición **2**
1.1 Estructura 5

2 Tipos **6**

3 Conjunto de instrucciones del lenguaje **8**

4 Gestión de errores **11**

5 Generación de Código **13**

1. Identificadores y ámbitos de definición

- **Declaración de variables simples sin valor inicial:** *tipo nombreVariable;*

  Por ejemplo: `int num;`

- **Declaración de variables simples con valor inicial:** *tipo nombreVariable = valor;*

  Por ejemplo: `int num = 0;`

- **Declaración de arrays sin valor inicial:** *tipo nombreArray[tamaño];*

  Por ejemplo: `int a[5];`

- **Declaración de arrays con valor inicial:** Nuestra forma de inicializar arrays será asignando los valores de cada elemento: *tipo nombreArray[N] = {valor1, ..., valorN};*
  Por ejemplo: `int a[5] = {0,1,2,3,1}` → `a=[0,1,2,3,1];`

  Cuando tengamos declarado un array (sea con valor inicial o no), para darle valor habrá que hacerlo de la misma forma: *nombreArray = {valor1, ..., valorN};*

  Por ejemplo: `a = {7,1,2,6,1}` → `a=[7,1,2,6,1];`

  Para acceder a los elementos del array será con *nombreArray[pos]* (siendo 0 la primera posición del array). De esta forma podemos cambiar el valor solo de un elemento del array u obtener el valor que tiene este.

  Por ejemplo: `a[0] = 4` → `a=[4,1,2,6,1];`

- **Declaración de varias dimensiones:** Será igual que la declaración de arrays, pero indicando en distintos corchetes el tamaño de cada dimensión. Este ha de ser una constante entera, no una expresión que dependa de variables.
  tipoDato nombreMatriz[dim1][dim2]...;

  Por ejemplo: `int myMatriz[2][2];`

  Para inicializar estos arrays será de la misma forma que antes, anidando los corchetes y sus valores:

  `int myMatriz[2][2] = {{1,2},{3,4}}`

  Para acceder a un elemento será igual, pero indicando las distintas posiciones dentro de los corchetes: *nombreArray[pos1][pos2]*

  Además, en estos casos cuando tengamos algo del estilo nombreArray[pos1] siendo el array en varias dimensiones, se devolverá un array de una dimensión menos que el original.

- **Bloques anidados:** Para anidar distintos bloques utilizaremos {...}

```
while(cond1){
   ...
   if(cond2){
      ...
   }
   ...
}
```

Cada bloque definido tiene su propio ámbito. Esto significa que podemos definir variables con el mismo nombre y tipo dentro de ambos. Cada una será visible en su ámbito. Si están anidados, al declarar una variable en el más externo también es visible en el interno. Sin embargo, si definimos otra variable con el mismo nombre en el interno, en este ámbito se impone la que acabamos de definir.
Por ejemplo:

```
while(cond1){
   int a = 0;
   if(cond2){
      a = a + 1;
      int a = 4
      a = a + 1;
   }      ...
}
```

En este caso la a declarada en el cuerpo del while es visible en el if hasta que se vuelve a declarar una nueva a. Por lo tanto, la primera operación daría $a = 1$ y la segunda $a = 5$

- **Registros (structs):** Los registros nos sirven para declarar nuevos tipos de variables que tengan a su vez distintas variables que actúen como 'atributos' suyos:

struct nombreStruct{
Declaración var1;
...
Declaración varN;
};

Por ejemplo:

```
struct persona{
   String nombre;
   String apellidos[2];
   String DNI;
   int edad;
};
```

`persona persona2;` → variable de tipo persona

El acceso a las distintos atributos de los structs se realizará: $nombreVariable.varStruct_i$
Por ejemplo: `persona1.nombre = David;`

De esta forma, se inicializarán las variables de tipo registro, inicializando cada variable del struct de forma individual.
Además, como los registros serán nuevos tipos podremos tener arrays de registros o variables de un registro que sean de tipo registro. También es posible tener registros declarados como variables dentro de otro registro.

- **Enumerados:** Los enumerados, al igual que los registros nos servirán para declarar nuevos tipos de variables en los que solo se pueden tomar una serie de valores constantes que declaramos cuando especificamos el enum.

  enum nombreEnum = { valor1, ..., valorN };

  Podemos tener una variable del tipo del enumerado, que ha de tener un valor de entre los definidos. No admitimos una inicialización con un entero k refiriéndose al valorK;
  Por ejemplo:

  ```
  enum diaLaboral = { LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES};
  diaSemana dia = LUNES;
  diaSemana dia3 = 3; → Error de tipado
  ```

- **Funciones:** Es necesario que el tipo ya se haya definido (enums o structs), o que sea un tipo básico: int, bool, char, String o arrays de estos mismos.

  tipo nombreFunción(param1,...,paramN){
  ...
  return c;
  }

  Siendo c una variable o una constante del tipo de la función. Si el tipo es void, no hay return dentro del cuerpo de la función. Destaca la función **main**, que va a ser nuestro punto inicial del programa. Es de tipo void y no recibe ningún parámetro de entrada.

  void nombreFunción(param1,...,paramN){
  ...
  }

  Los parámetros se pasan por valor. En el cuerpo de la función pueden ir distintas instrucciones y declaraciones de variables o arrays locales.
  Por ejemplo:

  ```
  int suma(int a, int b){
     int c = a + b;
     return c;
  }
  ```

```
void holaMundo(){
   //Mostrar por pantalla HolaMundo!
}
```

Estas variables son locales, es decir, solo son visibles en el ámbito de la función en la que sean definidas.

1.1. Estructura

Para configurar nuestro programa, es necesario seguir esta estructura:

1. Declaración de nuevos tipos enumerados y de registros. Pueden ir intercaladas entre ambas.
2. Declaración de las funciones del programa.
3. Declaración del main.

Por ejemplo:

enum nombreEnum = { valor1, ..., valorN };
struct nombreStruct{
Declaración var1;
...
Declaración varN;
};
int nombreFunción(param1,...,paramN){
...
return c
}
void main(){
...
return c
}

2. Tipos

- **Declaración explícita de tipos:** La declaración de una variable se hará poniendo su tipo antes del nombre de esta *tipo nombreVariable*. Por ejemplo `int num`, `bool stop`,...

- **Tipos básicos:** El lenguaje consta de 3 tipos básicos:

  1. **Enteros:** Este tipo estará especificado como `int nombreVariable`. El rango de valores que toma será el mismo que en C++ (valores entre -2,147,483,648 y 2,147,483,647). Las operaciones definidas para este tipo serán las operaciones básicas conocidas de suma resta, multiplicación, división entera y módulo para números enteros. Añadimos también la operación ++ y −−, cuya función es la misma que en C++. Para hacer comprobaciones con enteros, incluimos los operadores: `==`,`!=`,`>`,`<`,`>=` y `<=`.

  2. **Booleanos:** Este tipo estará especificado como `bool nombreVariable`. Toma los valores constantes `TRUE` y `FALSE`. A parte de la operación ==, se definen las siguientes:

     AND: `varBool1 && varBool2`
     OR: `varBool1 || varBool2`
     NOT: `!varBool`

&&	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

||	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

!	TRUE	FALSE
	FALSE	TRUE

  3. **Char:** Este tipo tomará como valores caracteres del teclado español. Deben ir declarados entre comillas simples. Estos valores serán exclusivamente:

     - Las letras del abecedario(con la ñ): `a, b, c, ..., z`
     - Vocales con tilde: `á, é, í, ó, ú`
     - Los números: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
     - Otro caracteres: `,|.|;|:|!|?|¿|(|)|{|}|[|])`

     La única operación que se podrá hacer con los char es la comprobación de ==.

  4. **String:** Cadena de caracteres. Debe declararse entre comillas dobles.

     ```
     String cadena = "Hola";
     ```

- **Tipos de usuario:** El usuario podrá declarar nuevos tipos declarando registros o declarando enums de la forma que hemos indicado en el apartado 1.

- **Tipo array:** Se permiten arrays de los tipos básicos presentados y de los posibles nuevos tipos definidos por el usuario. La sintaxis, forma de acceder a los elementos, etc. ya ha sido expuesta en el apartado 1.

- **Operadores infijos:** Primero se presentan los operadores para el tipo entero:

  1. **Suma**. Se representa mediante `+`. Prioridad 2. Asocia por la derecha.
  2. **Resta**. Se representa mediante `-`. Prioridad 2. Asocia por la derecha. También puede ser un operador unitario (-4). En este caso, no es asociativo.
  3. **Multiplicación**. Se representa mediante `*`. Prioridad 3. Asocia por la derecha.
  4. **División entera**. Se representa mediante `/`. Prioridad 3. Asocia por la derecha.
  5. **Módulo**. Se representa mediante `%`. Prioridad 4. Asocia por la derecha.
  6. **Operaciones de igualdad y orden**. Se representan mediantes los operadores descritos anteriormente (==,>,...). Todos ellos tienen prioridad 0 y no son asociativos.

  A continuación se muestran los del tipo bool:

  1. **Not**. Operador unario no asociativo. Prioridad 4
  2. **And**. Prioridad 0. Asocia por la derecha.
  3. **Or**. Prioridad 0. Asocia por la derecha.

  Para poder combinar 2 o más operadores con diferentes prioridades se incluyen los **paréntesis**: `( , )`. Siempre tiene que haber igual o más '(' que ')' mientras leemos de izquierda a derecha y al final debe haber el mismo nº de ambos.

- **Equivalencia estructural de tipos:** No permitimos añadir dos structs o enums con el mismo nombre. Por lo tanto, a pesar de que dos structs contengan el mismo nº de variables y sean del mismo tipo, seguirán siendo dos tipos distintos.

- **Comprobación de tipos:** Se realizará una comprobación estática de los tipos, es decir, se realiza en el momento de la compilación.

  - En cuanto a las llamadas de funciones, permitimos que dos funciones tengan el mismo nombre siempre que se diferencien en el nº de parámetros que recibe. Por lo tanto, es necesario comprobar cuántas se llaman como ella y ver con cuál cuadra.
  - Comprobamos que las condiciones del if y while son booleanas.
  - Comprobamos que todos los índices de los cases tienen el mismo tipo que la expresión del switch.
  - Comprobamos la corrección del tipo de las expresiones, extrayendo el tipo de las variables mediante el proceso de vinculación.
  - Comprobamos que el tipo de las expresiones para acceder a un array sean enteras. También comprobamos que los tipos de los elementos del array coincidan con el declarado.
  - No permitimos variables con el mismo nombre que structs o enums declarados.
  - Comprobamos los tipos de los atributos de los structs cuando tratamos de acceder a ellos.

3. Conjunto de instrucciones del lenguaje

- **Expresiones enteras:**
  - **Casos base:**
    1. *Constantes.* Cualquier constante entera.
    2. *Variables enteras.* `int a = 3; (se usa solo la a)`
    3. *Llamadas a funciones de tipo int.* `suma(a + b);`
  - **Caso Genérico:** Combinación de los casos base mediante los operadores aritméticos explicados anteriormente, de forma que entre dos operadores siempre habrá una expresión entera. Además, han de empezar y terminar siempre por expresiones enteras. Estas se pueden combinar con los paréntesis para las prioridades como explicamos previamente

    Ejemplo:
    ```
    int a = 2;
    int b = 1;
    a + 3 * (5 + b) + suma(a + 4) → 20
    ```

- **Expresiones booleanas:**
  - **Casos base:**
    1. *Constantes.* True o False.
    2. *Variables booleanas.* `bool b = True (se usa solo la b);`
    3. *Llamadas a funciones de tipo bool.* `esPar(a);` → `siendo esPar de tipo bool`
    4. *Comprobación de expresiones enteras*: Dadas 2 expresiones enteras, pueden compararse mediante los 6 operadores de orden e igualdad definidos en el apartado 2.
    5. *Comprobación de otros tipos*: Dadas 2 variables (del mismo tipo) ver si ambas tienen el mismo valor o no con `== y !=`
  - **Caso Genérico:** Combinación de los casos base mediante los operadores booleanos explicados anteriormente.

    Ejemplo:
    ```
    int a = 1;
    int b = 2;
    bool c = TRUE;
    char d = 's';
    char e = 'f';
    (c && esPar(b) && (a >= b)) || (e == d) → False
    ```

- **Instrucciones de asignación:** Las instrucciones de asignación serán de la forma
  nombreVariable = *x*

  Donde x es una expresión que tomará un valor del mismo tipo que la variable (en caso de int y bool pueden ser expresiones que le asignen el valor final de esta a la variable)

En el caso de los arrays se hará igual, es decir, se hace de la misma forma que las inicializaciones, pero sin poner el tipo al principio. También se pueden hacer asignaciones de solamente un elemento del array (todo esto está visto en el apartado 1).
Cuando indicamos la pos del elemento que queremos acceder, esta puede ser una expresión entera en vez de un valor entero.

- **Condicionales:** Si se verifica la condición entra por la rama del if. En caso contrario, se ejecuta el cuerpo del else (si es que existe pues este no es obligatorio).

```
if(cond){
   ...
}
else{
   ...
}end
```

- **Bucle:** Se emplea la sintaxis habitual del while. La condición puede incluir variables modificadas en el cuerpo del while; o estar formada por constantes de tal forma que en todas las vueltas siempre se verifique. Por lo tanto, es posible que aparezcan bucles infinitos, aunque debe ser el programador el encargado de evitarlos. Para ayudarles, se incluye una instrucción: **bartolo**. Debe ir en el cuerpo del bucle y finaliza su ejecución. El cuerpo del bucle se ejecutará completamente cada vez que se verifique la condición, que se comprobará antes de entrar en él y cada vez que este termine por si hay que salir o seguir ejecutándolo. Ejemplo:

```
while(cond){
   ...
   if(...){ → esto nos muestra un ejemplo, no es obligatorio que haya un if
      ...
      bartolo; → automáticamente se saldría del bucle
   }
   else{
      ...
   }end
   ...
}
```

- **Llamadas a funciones:** Siempre que tengamos un función previamente declarada, podemos hacer una llamada a esta, pasando el mismo nº de parámetros que tuviera esta sin decir el tipo de estos (aunque tiene coincidir el tipo de los parámetros uno a uno con los de la definición). Estos parámetros pueden ser constantes, expresiones o variables del tipo correspondiente.
  Los parámetros se pasarán siempre por copia, es decir, en memoria se realiza una copia del parámetro pasado y al terminar la función estas copias se eliminarán. Por tanto, los valores de los parámetros (si han sido modificados) se perderán, excepto el que devolvamos con el return en la función.
  La llamada de funciones de tipo `void` se pueden realizar llamando a la función simplemente: *nombreFuncion(param1, ..., paramN);*
  Sin embargo, para las funciones de otro tipo se tendrán que asignar sus llamadas a variables (inicializadas ya o no): *nombreVariable = nombreFuncion(param1, ..., paramN);*
  Esto más correctamente es un caso de una asignación en la que la llamada a la función es una expresión básica.
  Por ejemplo (con la función suma declarada en el apartado 1):

```
int d = 0;
int a = 3;
d = suma(a, 6); → d vale ahora 9
holaMundo(); → se muestra por pantalla Hola Mundo!
```

- **Switch:** Será una especie de instrucción condicional en la que se utilizará una variable o expresión y tendremos distintas ramas para cada uno de los posibles valores de esta. Estas ramas se indicarán poniendo *case posibleValor: ...(Instrucciones).*
  Volveremos a usar la expresión **bartolo** para salir directamente de la instrucción switch si no queremos seguir analizando si podemos entrar en más ramas (normalmente serán ramas disjuntas por lo que sería conveniente usar esta instrucción al final de cada rama.
  Mostraremos su uso con un ejemplo:

```
enum dias = { LUNES, MARTES, JUEVES };
dias dia = LUNES;
int a = 2;
int b = 3;
int d;
switch(dia){
  case LUNES:
    d = a + b;
    bartolo;
  case MARTES:
    d = b;
    bartolo;
  case JUEVES:
    d = a;
    bartolo;
}
```

- **Print:** Instrucción para imprimir por pantalla una cadena de caracteres o el valor de una expresión entera, de tipo char o string. Usaremos \n como salto de línea en la cadena de caracteres.

  print("cadena de caracteres");
  print(expresión);

  Por ejemplo:

  ```
  print("Hola Mundo");
  int a = 3;
  print(a + 2); → imprime 5 por pantalla
  ```

4. Gestión de errores

El manejo de los errores es parte esencial del compilador. Debe ser capaz de detectarlos y notificarlos. Además, tiene que recuperarse de ellos sin perder demasiada información, pues en caso contrario solo notificaría el primer error al usuario en cada proceso. Esto resulta muy poco eficiente para el programador cuando tiene que solucionar varios errores.

- **Detección de errores:** A continuación se describen los posibles tipos de errores, así como algunos ejemplos que los ilustren:

  1. **Error sintáctico:** error de sintaxis o ausencia de caracteres necesarios para completar instrucciones:
     - Ausencia de ';' al final de declaraciones de variables, asignaciones y llamadas a funciones.
     - Ausencia de '(' o ')' en las expresiones tanto booleanas como aritméticas; en las condiciones del if, switch,while; y en la definición o llamada de funciones.
     - Ausencia de '{' o '}' en los cuerpos condicionales, while, switch y funciones.

       ```
       if (a==2){
         return 1;
       else{        → Error de sintaxis: Falta } al final de if
         return 3;
       }end
       ```

     - Ausencia de '[' o ']' en declaración y tratamiento de arrays.

- Ausencia de operadores en las expresiones. `4 (2+4)`.
- Palabra reservada mal deletreada: Int, swich, wile...
- If, switch, while sin la condición necesaria para implementarse.

```
while (){
...
}
```

- Else sin aparición previa del if.
- Switch sin case o sin condición asociada a cada case.
- Expresiones que conformar por sí mismas una instrucción. Es decir, tanto las aritméticas como las booleanas deben formar parte de una asignación, o de la condición de if,switch y while.

2. **Error de vinculación:** Se da cuando utilizamos variables que no han sido definidas previamente, o que no están visibles en el ámbito actual. En cuanto a las llamadas a funciones, tendremos error de vinculación cuando queramos referirnos a funciones que no existen. De igual modo sucede con los structs o enums. Además, si deseamos declarar el tipo de una función a partir de algún enum o struct, estos deben estar definidos anteriormente. Podemos referirnos a funciones que son declaradas a posteriori.
3. **Error de tipo:** Aparición de variables o expresiones cuyo tipo no encaja con el requerido.
   - Variables inicializadas con expresiones de otros tipos. `int a = TRUE`
   - Uso de variables en expresiones cuyo tipo no es el adecuado.
     ```
     int a=2;
     if (a)...;
     ```
   - Uso de expresiones booleanas o aritméticas donde no corresponden.
     ```
     int A[5]={...};
     A[5==3]=7;
     ```
   - Argumentos pasados a una función con tipo incorrecto. Llamadas a funciones cuyo tipo no concuerda con el requerido en la expresión.

- **Notificación de errores:** Se mostrará por pantalla una lista con todos los errores detectados, así como la fila en la que se encuentra y el tipo de cada uno de ellos. Es posible que un mismo error de origen a otros. Por ejemplo, intentar acceder a una posición de un array, que desataría uno de variable fuera de ámbito que no ha sido declarado como tal. En este caso nos gustaría recibir una indicación sobre como actuar.
- **Recuperación de errores:** Como ya se ha detallado, es necesaria una recuperación de errores en la que no se pierda excesiva información. A la hora de implementar la gestión de los errores sintácticos hemos tratado de reducir el nº de caracteres que no procesa el compilador. En cuanto a los errores de vinculación o de tipado, proseguimos con ambos procesos hasta llegar al final, independientemente del nº de fallos. Si en alguno de los dos se produce un error, se muestra por pantalla que ha fallado ese proceso.

5. Generación de Código

Hemos conseguido generar código funcional para el main, utilizando enteros o arrays multidimensionales de enteros. Hemos logrado generar código para if, while, print, asignaciones, declaraciones y todo lo referido a expresiones: accesos de variables o arrays; operaciones unarias y binarias; comprobaciones de orden e igualdad para condiciones de if. También hemos conseguido generar con éxito la diferencia de ámbitos existentes si aparecen bloques anidados. Hemos intentado implementar el switch a partir de las diapositivas ofrecidas, pero nos ha fallado y no hemos sido capaces de solucionarlo. Tampoco hemos implementado la inicialización de arrays presentadas en el apartado 1. Para inicializar un array habría que hacerlo yendo posición a posición.