

ASTERIX

Impertative programming language

Sergio Domínguez Cabrera y Enrique Carro Garrido



No, Obélix. Tú no tendrás la poción mágica.

ASTERIX EL GALO ¹

En el presente texto se realiza una descripción detallada del lenguaje de programación imperativo **Asterix**.

Introducción a Asterix

Asterix nace de la idea de ser un lenguaje de programación de paradigma imperativo, con tipado estático y portable. Los programas escritos en **Asterix** tienen que tener la extensión `.atx` y la codificación de los ficheros `.atx` es UTF-8. ¿Pero qué sería **Asterix** sin **Obelix**? El compañerismo de la serie se traslada al lenguaje de forma que el compilador del mismo recibe el nombre de este gran compañero.

Tipos básicos y variables

Por defecto, **Asterix** es capaz de distinguir 4 tipos básicos:

- **INTEGERIX**: Representa el subconjunto finito de los números enteros que pueden ser expresados con 4 bytes. Para declarar una variable con este tipo hay que usar la palabra reservada `intix`.
- **BOOLEANIX**: Representa los valores de lógica binaria, esto es dos valores, que en nuestro lenguaje representamos como `galo` (`true`) y `romano` (`false`). Empleamos un byte para

¹Esta es una frase famosa que se repite en toda la saga. Cuando era niño, Obélix cayó en un caldero de poción mágica, cosa que ocasionó efectos a perpetuidad. Sin embargo, eso no le impide a Obélix intentar obtener una poción mágica cada vez que se presenta la oportunidad.

almacenar estos valores. Para declarar una variable con este tipo hay que usar la palabra reservada `boolix`.

- **FLOATIX:** Representa el subconjunto finito de los números racionales que pueden ser expresados con 8 bytes. Para declarar una variable con este tipo hay que usar la palabra reservada `floatix`.

Asterix permite agrupar estos tipos en **VECTIX** que son listas ordenadas de elementos de un único tipo (que también pueden ser **VECTIX**). Para declarar una variable como **VECTIX** hay que usar la palabra reservada `vectix`. Para acceder al elemento `i` del `vectix` se usa el operador `[]`.

Asterix también soporta la creación de **POTS**² que son una agrupación de variables que pueden ser de tipos distintos (incluso pueden contener otras **POTS**). Para declarar una variable con este tipo hay que usar la palabra reservada `pot`.

Declaración de variables

Para declarar una variable hay que seguir el siguiente formato:

```
tipo identificador (= valor inicial);
```

Las variables dentro de un mismo ámbito no pueden tener el mismo identificador. **Asterix** permite también la creación de variables globales. Para declarar una variable como global, se usa la palabra reservada `global` y se escriben fuera de la función `main`.

Variables sin inicializar

Empleamos el siguiente ejemplo para ilustrar su uso cuando las declaramos sin inicializar:

```
intix num1;
floatix num2;
vectix<vectix<intix>[3]>[3] matriz;
```

Estas variables sin inicializar, nosotros las vamos a guardar con un valor. Los `intix` se inicializan con el valor inicial 0, los `floatix` con 0.0 y los `charix` con el carácter ASCII 'a'. **Asterix** no permite la creación de `enumeratix` sin valor inicial.

Variables inicializadas

Empleamos el siguiente ejemplo para visualizar la declaración de variables con valor inicial:

```
floatix num2 = 1.0;
vectix<intix>[3] numeros = {1,2,3};
```

Declaración de nuevos tipos

Asterix permite al programador la creación de tipos mediante la palabra reservada `datix` seguida del identificador que queremos darle al nuevo tipo y del nuevo tipo. Vemos en el siguiente ejemplo cómo se usaría:

²La marmita o *pot* en inglés hace referencia a la marmita a la que se cayó Obélix.

```
// Creamos un tipo de pares de enteros
pot par {
    intix der;
    intix izq;
}

datix parix par;

// Creamos el tipo matriz 3x3
datix matrix3 vectix<vectix<intix>[3]>[3];
```

Caracteres especiales

Asterix permite al programador añadir comentarios de una línea al código. Estos comentarios son ignorados en el proceso de compilación junto con los espacios, tabulaciones y saltos de línea. Los comentarios comienzan siempre con el identificador `//` y se ignora todo lo que haya escrito hasta el siguiente salto de línea. Como ya avanzamos en la sección anterior, empleamos el caracter `;` para separar las instrucciones del programa. Mostramos ahora unos ejemplos de comentario para ayudar a comprender su uso:

```
// Esto es un comentario
intix variable; // Esta variable representa ...
```

Operadores

Asterix cuenta con una serie de operadores binarios y unarios que facilitan la programación. Escribimos en la siguiente tabla los operadores, su nivel de prioridad y su asociatividad. Los operadores aritméticos solo pueden ser aplicados sobre dos objetos del mismo tipo (y devuelven otro elemento de ese tipo), es decir, no podemos operar objetos de diferente tipo. No obstante, estos operadores tienen las mismas cualidades en ambos tipos.

Funciones

Asterix permite dividir el trabajo que hace un programa en tareas más pequeñas separadas de la parte principal. Estas tareas son lo que conoceremos como pociones. Todo programa escrito en **Asterix** comienza su ejecución en la función `panoramix`³ y es ella la que llama al resto de procedimientos. Las funciones siguen el siguiente esquema:

```
potion nombre_funcion(arg1 : type1,..., argn : typen) -> typer {
    typer var;
    // Cuerpo de la función
    return var;
}
```

³Panorámix, el druida. Creador de la poción mágica, el hombre más sabio del pueblo. En el primer libro, tiene su casa al lado de un manantial, y su cabaña tiene un palomar y una gran chimenea. Su nombre proviene de panoramique (panorámico). La razón de este nombre para el main del programa es ser el creador de las pociones, que son las subrutinas en **Asterix**

| Prioridad | Operador | Asociatividad | Aridad | Descripción |
|-----------|----------|---------------|---------|-----------------------------------|
| 0 | | Derecha | Binario | Operador or lógico |
| 1 | && | Derecha | Binario | Operador and lógico |
| 2 | == | Derecha | Binario | Operador igual relacional |
| 2 | != | Derecha | Binario | Operador distinto relacional |
| 3 | <= | Derecha | Binario | Operador menor o igual relacional |
| 3 | >= | Derecha | Binario | Operador mayor o igual relacional |
| 3 | > | Derecha | Binario | Operador mayor relacional |
| 3 | < | Derecha | Binario | Operador menor relacional |
| 4 | + | Izquierda | Binario | Suma de dos elementos |
| 4 | - | Izquierda | Binario | Resta de dos elementos |
| 5 | * | Izquierda | Binario | Producto de dos elementos |
| 5 | / | Izquierda | Binario | División de dos elementos |
| 5 | % | Izquierda | Binario | Modulo de dos elementos |
| 6 | ^ | Derecha | Binario | Potencia de dos elementos |
| 6 | ! | No asociativo | Unario | Operador not lógico prefijo |
| 7 | . | Izquierda | Binario | Acceso a elemento de un pot |
| 7 | [*] | Izquierda | Binario | Acceso a elemento de un vectix |
| 8 | - | Izquierda | Unario | Resta de un elemento |
| 9 | (*) | Izquierda | Binario | Denota precedencia |

```

potion nombre_procedimiento(arg1 : type1,..., argn : typen) {
    // Cuerpo del procedimiento
}

```

Todas las funciones se declaran con la palabra reservada **poc**. La flecha (\rightarrow) junto con **typer** indican el tipo del valor que devuelve la función, en su ausencia, significa que la función no devuelve ningún valor. Las variables **argi** son los parámetros que recibe la función y vienen escritas junto con su tipo **typei**. Si estamos pasando ese argumento por referencia, escribimos **&arg**.

Llamadas a función

Habíamos definido las funciones para dividir el programa en subrutinas que se pueden llamar desde el **panoramix** o desde otra subrutina.

Las funciones se pueden llamar desde cualquier sitio, teniendo en cuenta que deben ser coherentes con su tipo.

Para llamar a una función utilizaremos el siguiente formato:

```
nombre_funcion(arg1,arg2,...,argn)
```

Consideraciones a tener en cuenta:

- Se puede definir una función dentro de otra. Hay que tener en cuenta que se ejecutará primero la función más interna.
- Los argumentos de la llamada de la función, deben corresponderse tanto en tipo, número y orden en el que aparecen.

Instrucciones

En esta sección presentamos las instrucciones del lenguaje. Algunas de estas instrucciones dependen de una condición que ha de ser una sentencia que devuelva un valor booleano, y esta puede ser el valor lógico true (galo), si la condición se cumple, o romano si esta no se cumple (false). También puede contener el nombre de una variable booleano, y el valor de la expresión dependerá de su contenido. Se debe tener en cuenta que además de las variables también puede haber llamadas a funciones que devuelvan un valor.

Instrucciones condicionales

Asterix admite la estructura de bifurcación condicional, para determinar que acciones tomar dada o no cierta condición. La sintaxis que sigue la estructura if-then-else es la siguiente:

```
if ( condición ) {
    // Acciones a realizar si se cumple la condición.
}
else {
    // Acciones a realizar si no se cumple la condición.
}

// If sin else
if (condición) {
    // Acciones a realizar si se cumple la condición.
}
```

Para poder diferenciar unas con otras se obliga que se pongan llaves para indicar el ámbito de cada estructura.

Bucle while

El bucle while es un ciclo repetitivo basado en los resultados de una expresión lógica. El propósito es repetir un bloque de código mientras una condición se mantenga verdadera. La sintaxis que sigue esta estructura es:

```
while ( condición ) {
    // Acciones a realizar mientras se cumpla la condición.
}
```

Bucle for

Asterix soporta la estructura for que nos permite recorrer los elementos de un vector y es la siguiente:

```
for (type c : vect<type>) {
    // Bloque de instrucciones
}
```

En este ejemplo realizamos una vez el bloque de instrucciones por cada elemento del vector que instanciamos como c en cada iteración.

Más instrucciones del lenguaje

Asterix viene con más instrucciones por defecto:

- **return X**: Es obligatoria al final de las pociones que devuelven un valor de un cierto tipo. Indica el valor que se devuelve.

Entrada y salida

Un programa **Asterix** puede realizar operaciones de entrada y salida. En esta sección supondremos que la entrada proviene del teclado y que las salidas se envían a la pantalla de un terminal.

Entrada con *tabellae*⁴

tabellae es el flujo de entrada estándar. La entrada se hace por teclado y para que el programa interprete la salida como un tipo concreto **Asterix** tiene distintas instrucciones que las reconoce por separado.

El formato general de todas esas instrucciones son:

```
tabellae( Variable a leer );
```

El tipo de la variable puede ser **boolix**, **intix**, o **floatix**. Lo que hace la sentencia anterior es leer un dato introducido por teclado, interpretarlo con el tipo correspondiente y almacenarlo en la variable.

Salida con *stilus*⁵

Los valores de variables se pueden enviar a la pantalla empleando *stilus*. El formato general es:

```
stilus(Valor de una variable);
```

El valor de variables pueden ser constantes o variables de tipo **intix**, **floatix** y **boolix**

```
intix x = 5;  
stilus (x); // Saca por consola el valor 5
```

Gestión de errores

Asterix proporciona una gestión de errores básica indicando la línea y el motivo del error. El compilador intentará seguir con el análisis sintántico y semántico buscando posibles errores posteriores.

⁴Hace referencia a las *tabellae ceratae* en latín o tablillas de cera, que eran la herramienta que utilizaban los romanos para escribir notas o textos no muy largos. Entendemos que la *tabellae* es realmente la consola y que realmente leemos los datos de la *tabellae*.

⁵La herramienta que utilizaban los romanos para escribir en estas *tabellae*. Eran una especie de punzones hechos de hierro, de bronce o de hueso, con un extremo puntiagudo y el otro plano o redondeado. Se escribía sobre la capa de cera con el extremo puntiagudo y con el otro, si era necesario, se borraba y la tableta quedaba lista para volver a escribir. Es el antecedente directo de los lápices actuales que llevan una goma de borrar fijada en un extremo.

Ejemplos de código

Veamos ahora una serie de ejemplos de código que ilustren el uso del lenguaje Asterix . Empezamos con el ejemplo del bubblesort.

```
potion swap(xp : &intix, yp : &intix) {
    intix temp = xp;
    xp = yp;
    yp = temp;
}

// bubble sort
potion bubbleSort(v : &vectix<intix>[5]) {
    intix i = 0;
    intix j = 0;
    whileix (i < 5-1) {
        whileix (j < 5-i-1) {
            if (v[j] > v[j+1]) {
                swap(v[j], v[j+1]);
            }
            j=j--1;
        }
        i=i+1;
    }
}

// Función para imprimir el array
potion printArray(v : vectix<intix>[5]) {
    forix (intix i : v) {
        stilus(i);
    }
}

potion panoramix() -> intix {
    vectix<intix>[5] v = {-64, 34, 25, 12, 22};
    bubbleSort(v);
    printArray(v);
    return 0;
}
```

Veamos ahora como sería un programa que multiplica dos matrices 3x3.

```
// Programa para multiplicar dos matrices, una es una matriz global, la matriz
// A y la otra la recibimos por consola, la matriz B.
```

```
datix matrix vectix<vectix<intix>[3]>[3];
matrix A = {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}};

potion readmatrix(B : &matrix) {
    forix (vectix<intix>[3] fila : B) {
        forix (intix elem : fila) {
            tabellae(elem);
        }
    }
}
```

```

}

potion printmatrix(C : matrix) {
    forix (vectix<intix>[3] fila : B) {
        forix (intix elem : fila) {
            stilus(elem);
        }
    }
}

potion isInside(valor : intix, C : matrix) -> boolix {
    boolix pertenece = romano;
    forix (vectix<intix>[3] fila : B) {
        forix (intix elem : fila) {
            if (elem == valor) {
                pertenece = galo;
            }
        }
    }
    return pertenece;
}

potion multmatrix(A : matrix, B : matrix, C : &matrix) {
    intix i = 0;
    intix j = 0;
    intix k = 0;
    whileix (i < 3) {
        whileix (j < 3) {
            whileix (k < 3) {
                C[i][j] = C[i][j] + A[i][k]*B[k][j];
                k = k+1;
            }
            j = j+1;
        }
        i = i+1;
    }
}

potion panoramix() -> intix {
    matrix B;
    matrix C; // C = A * B

    // Leemos la matriz B por consola
    readmatrix(B);

    // Calculamos el producto
    multmatrix(A, B, C);

    // Escribimos el resultado por consola
    printmatrix(C);

    // Vemos si el 4 esta en la matriz C
    stilus(isInside(4,C));
}

```



```
    return 0;  
}
```