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.
- Charix: Representa el conjunto de caracteres de la codificación UTF-8. Para declarar una variable con este tipo hay que usar la palabra reservada charix.

[1] Esta 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.

- STRINGIX: Representa una lista de tamaño variable de charix. Para declarar una variable con este tipo hay que usar la palabra reservada stringix.
- 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 almacenar estos valores. Para declarar una variable con este tipo hay que usar la palabra reservada boolix.
- ENUMERATIX: Representa un tipo ordinal cuyo orden se indica por la disposición de los valores en la definición. Para declarar una variable con este tipo hay que usar la palabra reservada enumix.
- 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:

```
global intix numero_global;
intix num1;
floatix num2;
charix c;
stringix str;
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 caracter ASCII 'a'. `Asterix` no permite la creación de enumeratix sin valor inicial.

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

Variables inicializadas

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

```
global intix numero_global = 0;
floatix num2 = 1.0;
enumix dias_clase = {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES};
charix a = 'a';
stringix dia = "dia";
vectix<intix>[3] numeros = {1,2,3};
```

Declaración de nuevos tipos

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

```
// 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.

Prioridad	Operador	Asociatividad	Aridad	Descripción
0	,	Izquierda	Binario	Operador coma
1	=	Derecha	Binario	Operador de asignación
2	$\|\|$	Derecha	Binario	Operador or lógico
3	&&	Derecha	Binario	Operador and lógico
4	==	Derecha	Binario	Operador igual relacional
4	!=	Derecha	Binario	Operador distinto relacional
5	<=	Derecha	Binario	Operador menor o igual relacional
5	>=	Derecha	Binario	Operador mayor o igual relacional
5	>	Derecha	Binario	Operador mayor relacional
5	<	Derecha	Binario	Operador menor relacional
6	+	Izquierda	Binario	Suma de dos elementos
6	-	Izquierda	Binario	Resta de dos elementos
6	#	Derecha	Binario	Operador concatenación stringix
7	*	Izquierda	Binario	Producto de dos elementos
7	/	Izquierda	Binario	División de dos elementos
7	%	Izquierda	Binario	Modulo de dos elementos
7	$\wedge$	Derecha	Binario	Potencia de dos elementos
8	!	No asociativo	Unario	Operador not lógico prefijo
8	&	Asociativo	Unario	Paso de elemento por referencia
8	++	Asociativo	Unario	Incremento de prefijo (postfijo)
8	−−	Asociativo	Unario	Decremento de prefijo (postfijo)
9	.	Izquierda	Binario	Acceso a elemento de un pot
9	[*]	Izquierda	Binario	Acceso a elemento de un vectix
9	(*)	Izquierda	Binario	Denota precedencia

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;
}


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

Todas las funciones se declaran con la palabra reservada `poc`. La flecha (`->`) 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

[3]Panorámix, 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`

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. En caso contrario, deberá dar fallo de sintaxis.

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 booleanix, 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 booleanix, 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 pociones 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.
}
```

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

Bucle whilix

El bucle whilix 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:

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

Bucle forix

`Asterix` soporta la estructura forix, que nos permite ejecutar de manera iterativa un bloque de instrucciones, conociendo previamente un valor de inicio, un tamaño de paso y un valor final para el ciclo.

La sintaxis de esta instrucción viene expresada por el siguiente esquema:

```
forix (inicializacion; condicion; incremento) {
    // Bloque de instrucciones
}
```

La inicialización consiste en iniciar una variable que determina la condición de parada del bucle. Esta variable se irá incrementando con un cierto valor que denominaremos paso, que puede ser positivo, y por ello la variable se incrementará en cada iteración o puede ser negativo, por lo que irá decrementando en este caso.

Tanto la inicialización como la condición de parada se evalúan al principio de cada iteración y al final, el incremento.

Del mismo modo, `Asterix` permite otra sintaxis para el forix más segura a la hora de recorrer vectix y es la siguiente:

```
forix (type c : vectix<type>) {
    // Bloque de instrucciones
}
```

En este ejemplo realizamos una vez el bloque de instrucciones por cada elemento del vectix 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.
- `skip`: Instrucción vacía. El programa continua con la siguiente instrucción.

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 charix, 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 cadenas de caracteres que se expresen entre comillas, como por ejemplo,

```
stilus ("Por Tutatis");
```

o pueden ser variables de tipo intix y floatix,

```
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.

Ejemplos de código

No sería un lenguaje de programación como Tutatis[6] manda si no es capaz de recibir al programador con un "Hola mundo", aunque vamos a intercambiar esta frase por otra. ¿Adivinas cuál va a ser?

[4] Hace 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.

[5] La 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.

[6] **Teutates**, también llamado **Tutatis**, es la deidad de la unidad tribal masculina del panteón galo, según la antigua mitología celta. Hace referencia a la famosa frase que Ásterix repetía una y otra vez: "Por Tutatis"

```
potion panoramix() -> intix {
    stilus("Por Tutatis \n");
    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];
global matrix A = {{1 2 3}, {1 2 3}, {1, 2, 3}};

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

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

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

    // Escribimos el resultado por consola
    stilus("La matriz resultante de multiplicar A * B es: ");

    writematrix(C);

    return 0;
}

potion readmatrix(B : &matrix) {
    for (intix i = 0; i < 3; i++) {
        for (intix j = 0; j < 3; j++) {
            intix tmp;
            tabellae(tmp);
            B[i][j] = tmp;
        }
    }
}

potion writematrix(C : &matrix) {
    for (intix i = 0; i < 3; i++) {
        for (intix j = 0; j < 3; j++) {
            stilus(C[i][j]);
            stilus(" ");
        }
        stilus("\n");
    }
}
potion multmatrix(A : matrix, B : matrix) -> matrix {
    matrix C;
    for (intix i = 0; i < 3; i++) {
        for (intix j = 0; j < 3; j++) {
            for (intix k = 0; k < 3; k++) {
```

```
                C[i][j] = C[i][j] + A[i][k]*B[k][j];
            }
        }
    }
    return C;
}
```