JajaLANG

Mario Calvarro Marines
Beñat Pérez de Arenaza Eizaguirre

Introducción

Requisitos

En esa sección daremos los detalles sobre la sintaxis de nuestro lenguaje. La dividiremos en varias secciones siguiendo el enunciado de la práctica. En primer lugar, debemos recalcar que todas las sentencias del lenguaje acaban con punto y coma y los comentarios pueden ser unilínea (usando //) o multilínea (usando /* */). También es importante clarificar que la extensión de los ficheros de este lenguaje será «.jaja».

Identificadores y ámbitos de definición

A la hora de declarar una nueva variable debemos indicar el tipo de la misma y su identificador, acabando en punto y coma. Respecto a los arrays, permitiremos la creación de arrays de dimensión arbitraria cuyo tamaño puede venir dado en tiempo de ejecución por una variable o en compilación de forma constante. La sintaxis para declarar estos arrays será el tipo de los elementos del array, seguido del identificador del mismo y corchetes que abren y cierran. Cada pareja de corchetes indicará una dimensión más. Por ejemplo,

```
ent id1;
bin id2;
ent arr1[];
ent arr2[][];
```

Figura 1: Ejemplo de declaración de variables y arrays de una y dos dimensiones.

Los bloques anidados simplemente serán delimitados por llaves, por ejemplo,

```
si cond {
    mientras condicion {
        //Codigo
    }
}
```

Figura 2: Ejemplo de bloques anidados con un condicional y un bucle.

Las funciones se componen de cinco partes diferenciadas. Primero, declaramos que es una función a través de la palabra clave «diver» (que proviene de la palabra inglesa *fun* (*function*)). Tras esto, incluimos el nombre que se le da a la función seguido de los argumentos, separados por flechas. Estos argumentos tendrán la siguiente forma: identificador del parámetro y su tipo. Por defecto, el paso de parámetros será por valor, en caso de que sea por referencia, se deberá añadir el símbolo «&». Finalmente, a través de una flecha indicamos el tipo de retorno de la función y entre corchetes el cuerpo de la misma. El tipo de retorno es opcional en el caso de que únicamente se modifique el estado del programa. Por ejemplo,

```
diver jubilo (a: ent -> &b: ent) -> ent {
    //Codigo
}
```

Figura 3: Ejemplo de la función *jubilo*, que recibe dos enteros (uno de ellos por referencia) y devuelve otro.

Hemos decidido incluir en nuestro lenguaje tanto punteros como registros. La declaración de los punteros la realizaremos similarmente a la de las variables, pero incluyendo el símbolo (@) entre el tipo y el identificador. Respecto a los registros se hace de manera similar a las variables habituales, pero añadiendo la palabra clave «registro» en primer lugar. Se asume que este tipo de datos ha sido definido correctamente con anterioridad. Estos datos se definen de la siguiente manera: en primer lugar, utilizamos la palabra clave «registro», seguido del nombre de este nuevo tipo. Tras esto, iniciamos un bloque anidado con los campos del registro y su tipo, separados por comas (el último también puede tener coma al final de forma opcional). Por ejemplo,

```
registro datos {
    a: ent,
    b: bin,
}

entero @ p;
registro datos reg;
```

Figura 4: Ejemplo de la declaración de un puntero a un entero y de un registro con dos campos.

Para importar código procedente de otros ficheros utilizamos la palabra clave «#traficar» seguido de la localización del otro fichero. Por ejemplo,

```
#traficar ruta/a/fichero.jaja
```

Figura 5: Ejemplo de la declaración de un puntero a un entero y de un registro con dos campos.

Tipos

Como ya hemos indicado anteriormente, las variables tienen que venir declaradas de forma explícita y su tipado es estático. Los tipos predefinidos del lenguaje serán los «enteros» y los «binarios» (booleanos). La lista de operadores predefinidos será la siguiente (el orden en que aparecen en la lista indica la prioridad siendo el primero el más prioritario, pero siendo los paréntesis lo más prioritario de todo).

- Operadores aritméticos:
    1. Potenciación (^).
    2. Producto (*), división (/) y módulo (%).
    3. Suma (*) y división (-).
- Operadores relacionales (todos tienen la misma prioridad):

  Comparaciones: ==, !=, >, <, >=, <=.
- Operadores lógicos:

1. Negación lógica: !.
2. ≪Y≫ lógico: &&.
3. ≪O≫ inclusivo lógico: ||.

La asociatividad de los distintos operadores será la habitual.

El tipo array será como el indicado en la anterior sección.

El usuario podrá definir sus propios tipos haciendo un alias de tipos compuestos (registros o arrays). La estructura para realizar un alias será la siguiente: la palabra clave ≪incognito≫ seguido del alias y un igual. Tras esto, se escribirá la expresión que se abrevia. Por ejemplo:

```
incognito matriz = ent[][];
```

Figura 6: Ejemplo de uso del alias.

Conjunto de instrucciones del lenguaje

En nuestro lenguaje habrá presentes multitud de instrucciones de asignación dependiendo del tipo de la variable (simples, arrays o registros). Todas ellas tendrán en común el uso del operador igual (=) y la siguiente estructura: en el lado izquierdo de la asignación tendremos una declaración de una variable o su identificador y en el derecho una expresión con un valor. Las expresiones posibles son las aritméticas y booleanas habituales (con los operadores anteriormente definidos), pero también hemos decidido incorporar el operador ternario (?) que evalúa una condición y dependiendo de esto asigna un valor u otro (que vienen dados, a su vez, por expresiones).

Los arrays serán asignados elemento a elemento, separando los valores dados por comas y todo ello entre corchetes. Por otro lado, los registros se asignarán valor a valor, pero de manera recursiva, cada campo individualmente. Por último, las asignaciones pueden combinarse con operadores para realizarse sobre sí mismas. Para asignar a un array una cantidad de memoria sin inicializar simplemente escribimos el tipo de los elementos seguido de, entre corchetes, el número de elementos. Para los registros inicializados sin valor inicial simplemente escribimos el tipo del registro. Por ejemplo,

```
ent a = 5;
a = cond? 3 - 1 : 2 + 2;      //Si se cumple, 2. En caso contrario, 4
a += 5;                       //a = a + 5

bin b = true;
bin c = !b;
c |= b                        //c = b || c

ent arr1[] = [2, 3, 4];
arr1 = [3, 2, 1];
ent arr2[] = ent[3];          //Array de size 3, pero sin valores

registro datos reg1 = {a = 2 + 1, b = cond? true : b || c,};
reg1 = registro datos         //Reservar espacio nuevo
```

Figura 7: Ejemplos de asignaciones.

La ejecución condicional en este lenguaje tendrá como palabras claves ≪si≫ y ≪sino≫. Estos condicionales tendrán n ramas siguiendo la siguiente estructura: empezamos con la palabra clave ≪si≫ seguida de una expresión condicional que se evalúa a un ≪binario≫ y un bloque anidado

de código (que se ejecutará si el «binario» se evalúa a 1. Tras esto, le seguirán n bloques que empezarán con la palabra clave «sino» y, opcionalmente, una condición y su correspondiente bloque anidado de código. Si no existe condición, se interpretará como un «en caso contrario» y se ejecutará si lo anteriores no lo han hecho. Por ejemplo,

```
si true || false || (3 != 2) {
    //Codigo
} sino false && true {
    //Codigo
} sino {
    //Codigo
}
```

Figura 8: Ejemplo de la ejecución condicional.

Hemos decidido incluir dos tipos de bucles que nombramos por «mientras» y «para» (provenientes de *while* y *for*). La sintaxis que sigue «mientras» es: palabra clave seguida de una expresión condicional y el bloque de código que se ejecuta mientras se cumpla la condición. Por otro lado, el bucle «para» es: palabra clave, la asignación de una nueva variable (que no es necesario declarar de forma explícita) a su valor inicial, una flecha y el valor final (no inclusivo) que tendrá la variable. Por ejemplo,

```
mientras a || b && !c {
    //Codigo

    para i = 0 -> 20 {
        //Codigo anidado
    }
}
```

Figura 9: Ejemplo de bucles (anidados).

Gestión de errores

Para la gestión de errores, el compilador simplemente imprimirá el tipo de error que se ha dado y su localización en el fichero de texto. No habrá recuperación de errores).