JajaLANG

Mario Calvarro Marines
Beñat Pérez de Arenaza Eizaguirre

Introducción

En este documento se presenta una especificación de la sintaxis del lenguaje que vamos a crear. Hemos decidido llamarlo *jajaLang* debido a que será un lenguaje de ≪juguete≫ (¡y muy divertido de programar!).

Por otra parte, las principales influencias a la hora de decidir como va a ser la sintaxis han sido *C*, *C++* y *Rust*. Simplemente eran los lenguajes que mejor manejamos y que en mayor estima tenemos.

A continuación presentamos la especificación detallada del lenguaje, pero consideramos que antes debemos aclarar un par de cuestiones. En primer lugar, los ficheros de este lenguaje tendrán la extensión *.jaja*. A su vez, los comentarios podrán ser de una sola línea (usando //) o de múltiples (usando /* */). Todas las sentencias acaban en punto y coma (;).

Requisitos

En esa sección daremos los detalles sobre la sintaxis de nuestro lenguaje. Siguiendo de manera aproximada el enunciado dividiremos las secciones de este apartado.

Léxico

Veamos en primer lugar el léxico, es decir, los *tokens* que manejaremos en nuestro lenguaje. Únicamente se manejan símbolos ASCII:

Palabras clave

Listado de las palabras clave del lenguaje:

- `ent`
- `bin`
- `si`
- `mientras`
- `diver`
- `registro`
- `incognito`

Identificadores

Los identificadores de las variables serán una letra o un guion bajo seguido de una secuencia de letras y números.

Espacios

Los espacios o secuencia nulas en nuestro lenguaje serán los espacio propiamente dichos (‘ ’), saltos de línea (‘`\n`’), tabuladores (‘`\t`‘), *carriage return* (‘`\r`’) y el *backspace* (‘`\b`’).

Identificadores y ámbitos de definición

Veamos la declaración de los distintos tipos de variables.

Variables simples

A la hora de declarar una nueva variable debemos indicar el tipo de la misma y su identificador.

```
tipo id;
```

Figura 1: Declaración de una variable simple.

También es posible declarar múltiples variables del mismo tipo, separándolas por comas:

```
tipo id1, id2;
```

Figura 2: Declaración de múltiples variables.

Arrays

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,

```
tipo arr[]...[];
```

Figura 3: Declaración de arrays de dimensión arbitraria.

Registros y punteros

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, pero añadiendo la palabra clave `registro` en primer lugar. 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).

```
//Registros
registro datos {
    atributo11, atributo12, ...: tipo1,
    atributo21, atributo22, ...: tipo2,
    ...
} id1, id2, ...;

//Punteros
tipo @ id;
```

Figura 4: Declaración de variables registro y de punteros.

Bloques anidados

Los bloques anidados simplemente serán delimitados por llaves.

```
{
    {
        //Codigo
    }
    {
        //Codigo
    }
}
```

Figura 5: Bloque con otros dos bloques anidados.

Funciones

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 `&` antes del identificador. Finalmente, a través de una flecha indicamos el tipo de retorno de la función y un bloque anidado con el cuerpo de la función. El tipo de retorno es opcional en el caso de que únicamente se modifique el estado del programa. Por ejemplo,

```
diver id1 (par1: tipo1 -> par2: tipo2 -> ... ) {
    //Codigo
}
diver id2 (&par1: tipo1 -> par2: tipo2 -> ... ) -> tipo {
    //Codigo
}
```

Figura 6: Declaración de funciones con y sin tipo de retorno y con parámetros por valor y por referencia.

Importación de código

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 7: Importación de código procedente de otro fichero.

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» de 32 bits (que declararemos con la palabra clave `ent`) y los «binarios» (booleanos, con palabra clave `bin`). La lista de operadores predefinidos será la siguiente.

- Operadores aritméticos:
  1. Potenciación: `^`.
  2. Producto (`*`), división (`/`) y módulo (`%`).
  3. Suma (`+`) y división (`-`).
- Operadores relacionales: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Operadores lógicos:
  1. Negación lógica: `!`.
  2. «Y» lógico: `&&`.
  3. «O» inclusivo lógico: `||`.
- Otros operadores:
  1. Acceso a atributos de un registro: `.`
  2. Acceso a un elemento de un array: `[]`
  3. Llamada a una función (identificador de la función a la izquierda, parámetros dentro): `()`
  4. Opuesto de un entero: `-`

5. Acceso a la dirección de una variable: `&`

6. Acceso al valor apuntado por un puntero: `@`

Por último, la asignación será `=` y se podrá combinar con las operaciones aritméticas y de lógicas.

Operador	Prioridad	Asociatividad
`[] .`	0	Izquierda
`()`	1	Izquierda
`-` unario	2	Derecha
`& @`	3	-
`* / %`	4	Izquierda
`+ -`	5	Izquierda
`< > <= >=`	6	Izquierda
`== !=`	7	Izquierda
`&&`	8	Izquierda
`||`	9	Izquierda
`= += ...`	10	Derecha

Cuadro 1: Tabla con los distintos operadores, su prioridad y su asociatividad.

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 8: 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 9: 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 10: 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 11: 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).