JajaLANG

Mario Calvarro Marines
Beñat Pérez de Arenaza Eizaguirre

1. 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 cómo va a ser la sintaxis han sido *C*, *C++* y *Rust*. Simplemente eran los lenguajes que mejor manejamos y que en mayor estima tenemos.

Todo programa deberá tener, si se quiere ejecutar y no usar como una librería, una función `tronco` que será el punto de inicio (como en un árbol).

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, todas las sentencias acaban en punto y coma (`;`).

2. Requisitos

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

2.1. Léxico

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

2.1.1. Palabras clave

Listado de las palabras clave del lenguaje:

- `ent`
- `bin`
- `facto`
- `fake`
- `si`
- `sino`

- `mientras`
- `para`
- `diver`
- `registro`
- `nulo`
- `incognito`
- `devuelve`
- `#traficar`
- `como`
- `vector`
- `nuevo`

2.1.2. Identificadores

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

2.1.3. 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`’).

2.1.4. Comentarios

Los comentarios en este lenguaje podrán ser de una sola línea (`//`) o ocupar múltiples (se abren con `/*` y se cierran con `*/`).

2.2. Identificadores y ámbitos de definición

Veamos la declaración de los distintos tipos de variables.

2.2.1. Variables simples

A la hora de declarar una nueva variable debemos indicar el tipo de la misma y su identificador. También es posible declarar múltiples variables del mismo tipo, separándolas por comas.

```
tipo id;
```

Figura 1: Declaración de una variable simple.

```
tipo id1, id2...;
```

Figura 2: Declaración de múltiples variables.

2.2.2. Arrays

Respecto a los arrays, permitiremos crear arrays de un tipo genérico, pero tamaño dado en tipo de compilación mediante la palabra clave `vector`. Con esto permitiremos crear arrays de un número genérico de dimensiones (simplemente en tipo ponemos otro array).

```
vector(tipo, tam) arr;
```

Figura 3: Declaración de arrays.

2.2.3. 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 a los arrays. Primero, utilizamos la palabra clave `registro`. Tras esto, iniciamos un bloque anidado con los campos del registro, separados por comas (el último también puede tener coma al final de forma opcional), y, separado por punto y coma, su tipo.

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

//Punteros
tipo @ id;
```

Figura 4: Declaración de variables registro y de punteros.

2.2.4. Bloques anidados

Los bloques anidados simplemente serán delimitados por llaves.

2.2.5. 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.

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

Figura 5: Bloque con otros dos bloques anidados.

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. En el caso de que queramos devolver un valor, utilizaremos la sentencia `devuelve valor;` que, además, hará que la ejecución del programa vuelva al lugar desde el que se llamó a la función.

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

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

2.2.6. Importación de código

Para importar código procedente de otros ficheros utilizamos la palabra clave `#traficar` seguido de la localización del otro fichero. Tras esto, añadimos la palabra clave `como` y el alias del *namespace* del fichero importado (para evitar la colisión de nombres). Para llamar a las funciones y acceder

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

Figura 7: Importación de código procedente de otro fichero.

a las variables declaradas en este otro fichero, usaremos el operador `::` precedido del identificador del *namespace* y seguido de aquello a lo que queremos acceder.

2.3. 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 resta (`-`).
- Operadores relacionales: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Operadores lógicos:
  1. Negación lógica: `!`
  2. «Y» lógico: `&&`
  3. «O» inclusivo lógico: `||`
- Otros operadores:
  - Acceso a atributos de un registro: `.`
  - Acceso a un elemento de un array: `[]`
  - Llamada a una función (identificador de la función a la izquierda, parámetros dentro): `()`
  - Opuesto de un entero: `-`
  - Acceso a la dirección de una variable: `&`
  - Acceso al valor apuntado por un puntero: `@`

  Por último, la asignación será `=` y se podrá combinar con las operaciones aritméticas y 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 el tipo que se abrevia.

```
incognito alias = tipo;
```

Figura 8: Declaración de un nuevo alias.

2.4. 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 asignación (`=`) y la siguiente estructura: en el lado izquierdo de la asignación tendremos una declaración de una variable o una expresión de acceso a una variable, array o registro y en el derecho una expresión con un valor. Las expresiones posibles son las aritméticas y booleanas habituales (con los operadores anteriormente definidos).

2.4.1. Variables simples

La asignación de variables simples simplemente se realizará con el operador igual. En el lado izquierdo de la operación se encontrará el identificador de la variable y en el derecho una expresión del tipo de la variable (aritmética o lógica). A su vez, será posible combinar las distintas declara-

```
id = expr;
```

Figura 9: Asignación de un valor a una variable.

ciones vistas anteriormente con la asignación para declarar y dar valor a la variable en una sola línea. Todas las variables se inicializarán implícitamente a `0`, en el caso de los enteros, `fake`, en el

```
tipo id = expr_tipo;
tipo id1 = expr_tipo, id2, id3 = expr_tipo...;
```

Figura 10: Declaración y asignación de múltiples variables de forma simultánea. Cabe destacar que, en este caso, la variable con identificador `id2` no tiene valor asignado.

caso de los binarios, o a `nulo`, en el caso de los punteros.

2.4.2. Arrays

Los arrays serán asignados elemento a elemento, separando los valores dados por comas y todo ello entre corchetes. 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. De nuevo, podemos declarar una variable de tipo array y, simultáneamente, asignarle un valor.

2.4.3. Registros

Por otro lado, los registros se asignarán valor a valor, pero de manera recursiva, cada campo individualmente y, de nuevo, se podrá declarar y asignar a la vez. Si simplemente queremos reservar memoria, la asignación se hará de manera implícita en la declaración.

```
//Asignacion a un array la lista de elementos eli
//Los elementos pueden ser, a su vez, arrays
arr = [el1, ..., eln];

//Declaracion y asignacion
vector(tipo, n) arr = [el1, ..., eln];
```

Figura 11: Asignación de valores a los arrays.

```
//Los elementos pueden ser, a su vez, registros o arrays recursivamente
reg = {
    atributo11 = expr11,
    atributo12 = expr12,
    ...
    atributo21 = expr21,
    atributo22 = expr22,
    ...
};
```

Figura 12: Asignación de valores a los registros.

```
//Declaracion y asignacion
registro {
    atributo11, atributo12, ...: tipo1,
    atributo21, atributo22, ...: tipo2,
    ...
} reg = {
    atributo11 = expr11,
    atributo12 = expr12,
    ...
    atributo21 = expr21,
    atributo22 = expr22,
    ...
};
```

Figura 13: Declaración y asignación de valores a los registros.

2.4.4. Ejecución condicional

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 los anteriores no lo han hecho.

```
si expr_cond1 {
    //Codigo1
}
sino expr_cond2 {
    //Codigo2
}
//Mas ramas condicionales
//...
sino {
    //CodigoN
}
```

Figura 14: Ejecución condicional.

2.4.5. Bucles

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 entera (que no es necesario declarar de forma explícita) a su valor inicial, una flecha y el valor final (no inclusivo) que tendrá la variable.

```
mientras expr_cond {
    //Codigo
}

para i = expr_arit_ini -> expr_arit_fin {
    //Codigo
}
```

Figura 15: Ejecución iterativa.

2.4.6. Entrada/Salida

Para realizar la entrada y la salida, el lenguaje contará con una serie de funciones predefinidas para permitirlo. Estas simplemente permitirán la lectura/escritura de enteros y binarios. En caso

de que se intente leer o escribir algo que no coincide con la función llamada, se producirá un error.

```
leerEnt();
leerBin();
escribirEnt(num);
escribirBin(var);
```

Figura 16: Funciones de lectura y escritura.

2.4.7. Memoria dinámica

Para reservar memoria localizada en el *heap* y luego liberarla hemos decidido la palabra clave `nuevo` para la reserva y la función `liberar` para liberarla.

Esta reserva consistirá simplemente en una expresión utilizando la palabra clave `nuevo` seguida del tipo que queremos reservar (por ejemplo, para reservar memoria dinámica para un entero sería `nuevo ent`). Esto hará que se reserve en el *heap* la cantidad de memoria que ocupe este tipo. La liberación consiste simplemente en llamar a `liberar` y como parámetro el puntero que señala el inicio de la región de memoria reservada anteriormente.

```
//Reserva de memoria
tipo @ punt = nuevo tipo;

//Liberar memoria
liberar(punt);
```

Figura 17: Funciones de memoria dinámica.

2.4.8. Composición

Aunque ya se haya dicho de manera implícita, la composición de instrucciones en este lenguaje se hará mediante el símbolo `;`.

2.5. Expresiones

La expresiones de nuestro lenguaje se dividirán en aquellas que sean aritméticas (es decir, que su valor sea un entero) y lógicas (con valor binario). Serán las siguientes:

- **Constantes**: Serán números enteros habituales (aritméticas) o las palabras clave `facto` (del latín *factum*, hecho) y `fake` (lógicas).
- **Variables**: Identificadores de las variables inicializadas anteriormente.
- **Operadores infijos**: Expresiones compuestas por dos operandos y, entre estos, uno de los operadores definidos anteriormente.

- **Llamadas a funciones**: Siguiendo la definición del operador `()` especificado anteriormente, serán simplemente el identificador de la función a llamar y, entre paréntesis, los argumentos con los que se la llama.

Las expresiones podrán ser, obviamente, una combinación de todas estas y se podrán entender semánticamente como el valor que producen como resultado.

2.6. 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.

3. Ejemplos

En esta sección presentaremos ejemplos de algunos programas habituales escritos en *jajaLang* para ilustrar las características del lenguaje y como afronta diversas situaciones.

3.1. Fibonacci

Programa que calcula los primeros 10 elementos de la serie de Fibonacci, de forma recursiva, y los muestra por pantalla.

```
diver fibonacci(r: ent) -> ent {
    si c == 0 {
        devuelve 1;
    }
    si c == 1 {
        devuelve 1;
    }
    devuelve fibonacci(c-1) + fibonacci(c-2);
}

diver tronco() {
    para i = 0 -> 10 {
        escribirEntero(fibonacci(i));
    }
}
```

Figura 18: Programa que calcula la serie de Fibonacci.

3.2. Lectura/Escritura

Programa que lee un entero y un binario. Si el binario es `facto`, se imprimirá el entero. En caso contrario, no se hará nada.

```
diver tronco() {
    ent num = leerEnt();
    bin cond = leerBin();
    si cond {
        escribirEnt(num);
    }
}
```

Figura 19: Programa que lee un número y una condición que, si se cumple, hace que se imprima el entero.

3.3. Par/Impar

Programa que calcula si un número es par (`facto`) o impar (`fake`). Para ello se hace uso de una recursión mutua.

3.4. Registros y arrays

Programa que comprueba el funcionamiento de los registros y los arrays, el acceso a sus atributos/elementos y su modificación.

3.5. Paso por referencia y memoria dinámica

Programa que pasa por referencia una variable y utiliza memoria dinámica.

```
diver par(x: ent) -> bin {
    si x == 0 {
        devuelve facto;
    }
    sino x == 1 {
        devuelve fake;
    }
    sino {
        devuelve impar(x-1);
    }
}

diver impar(x: ent) -> bin {
    si x == 0 {
        devuelve fake;
    }
    sino x == 1 {
        devuelve facto;
    }
    sino {
        devuelve par(x-1);
    }
}

diver tronco() {
    escribeBin(impar(9));
    escribeBin(par(12));
}
```

Figura 20: Programa determina si un número es par o impar, usando recursión mutua entre dos funciones.

```
incognito dato = registro { arr: vector(ent, 3), };

diver tronco() {
    dato x = nuevo { arr = nuevo [1, 2, 3] };
    escribeEnt(x.arr[0]);
    x.arr[0] = 9;       //Asociativo por la izquierda
    escribeEnt(x.arr[0]);
}
```

Figura 21: Programa que trata con los atributos/elementos de los registros/arrays.

```
incognito dato_t = registro {
    atr1: ent @,
    atr2: vector(ent @, 2),
};

diver f(&a: ent) -> ent {
    devuelve @a;
}

diver tronco() {
    ent @ punt = nuevo ent;
    @punt = 10;
    ent b = f(punt);
    escribeEnt(b); //Tiene que dar 10
    liberar(punt);


    //Ejemplo complicado de punteros
    //registros y arrays con mem. dinamica
    dato_t @ d = nuevo dato_t;
    @d = {
        atr1 = nuevo ent,
        atr2 = [nuevo ent, nuevo ent],
    }
    @(@d).atr1 = 4;
    (@d).atr2[0] = 1;
    (@d).atr2[1] = 2;
}
```

Figura 22: Programa que contiene una función con un parámetro que se pasa por referencia y punteros que hacen uso de memoria dinámica.