MI LENGUAJE: KR

Rubén Gómez Blanco

Febrero 2022

1. Introducción al lenguaje

El lenguaje de programación KR, debe su nombre a las iniciales K, de key, y Rubén, su desarrollador. ¿Y por qué "key"? Se debe a que en este lenguaje, todo irá encapsulado en bloques de código, delimitados por { y }, y teniendo siempre un bloque principal, al que llamaremos MB (main block).

A continuación, pasemos a la especificación de la sintaxis del lenguaje KR, la cual dividiremos en siete bloques: Estructura del programa, Identificadores y ámbitos de definición, Tipos, Instrucciones, Gestión de errores, Ejemplo de un programa y Modificaciones respecto a la idea inicial.

2. Estructura del programa

Todo programa del lenguaje empieza por el bloque del programa (entre llaves) y se subdivide en 4 grandes bloques:

1. Definición de tipos con typedef.
2. Definición de structs y sus campos.
3. Definición de funciones.
4. Main Block o bloque principal donde se ejecuta el programa.

Los tres primeros bloques pueden estar vacíos. Cada uno de estos bloques van envueltos con sus respectivas llaves de esta forma:

{ ({Bloque1}) ({Bloque2}) ({Bloque3}) {MB} }

3. Identificadores y ámbitos de definición

Un identificador es una secuencia de caracteres que se usa para denotar:

1. Nombre de una variable (simple, constante, array, punteros o structs)

2. Funciones y sus parámetros

3. Tipos de usuario

La secuencia de caracteres que componen el identificador debe ser de la forma: el primer carácter de la secuencia debe ser una letra del abecedario (minúscula o mayúscula) y después, una secuencia de letras, números (dígitos del 0 al 9 por cada carácter) y el carácter '_'.

Caracteres para identificadores:
_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9

La sintaxis para definir **variables simples** es la siguiente:

tipo identificador;

En la siguiente sección del documento se especifican los tipos disponibles.
La sintaxis para la **definición de arrays** es la siguiente:

array (tam_array) tipo identificador;

Como array va a ser uno de nuestros tipos, se nos permitirá anidar arrays, que permite crear arrays de varias dimensiones con esta sintaxis:

array (tam_array) array (tam_array)...array (tam_array) tipo identificador;

Todas las variables se pueden inicializar en la misma linea en la que se definen, los enteros con números enteros o expresión que devuelva un número entero, booleanos con true/false o bien una expresión booleana, o arrays de este tipo de datos, como se puede ver en el ejemplo siguiente. ***IMPORTANTE*** Es importante destacar que solo se pueden inicializar con listas de enteros o booleanos arrays unidimensionales.

```
int num;        bool cond;      array (3) int lista;
int num=1;      bool cond=true; array int lista=[1,2,3,4,5];
array (3) array (2) bool lista_bool;
```

Como ya hemos mencionado con anterioridad, el lenguaje KR se divide en **bloques**, cuya sintaxis será: { bloque de código }. Todo programa en nuestro lenguaje, debe empezar por el bloque de programa, por tanto siempre se empezará por un { y acabará con un }, debe tener como último bloque el main block.

Para la **definición de funciones**, usaremos la palabra reservada "def", seguido del tipo que devuelve (puede ser void si no devuelve nada), del identificador de la función, de sus parámetros (con & si se pasa el parámetro por referencia, en el caso de pasarlo por valor no añadimos nada) y del bloque de código que

realiza la función a definir.***IMPORTANTE*** 1-En el lenguaje se permiten funciones recursivas. 2-El return deben en la última línea del bloque de código de la función, y solo puede haber uno. 3-Hace falta especificar el tamaño de los arrays cuando son multidimensionales, pero no cuando son unidimensionales. La sintaxis es de la forma:

def tipo/void identificador (tipo(&)identificador,...,tipo(&)identificador) bloque

En la imagen se puede ver un ejemplo:

```
int suma_deja_a_cero(int & x, int & y){
    int res=x+y;
    y=0;
    x=0;
    return res;
}
```

Por último, podremos definir también estructuras de datos (más conocido en otros lenguajes como **structs**) con la sintaxis siguiente:

struct identificador {conjunto de campos con sus tipos}

En la siguiente imagen se ve un ejemplo:

```
struct st{
    int x;
    int y;
    array (5) int lista;
}
```

Para acceder a los distintos campos de un struct, se accede con esta sintaxis:

iden_struct.campo (= valor);

IMPORTANTE Es importante destacar que el bloque de structs debe ser el segundo bloque de nuestro proyecto (el primero son los typedefs del usuario si los hubiera). Es importante destacar que se puede acceder a un campo de un struct que sea del tipo array, y acceder a posiciones de este array. Sin embargo, no se puede tener una lista de structs.

4. Tipos

Como ya hemos adelantado en la sección anterior, los **tipos** disponibles son números enteros y booleanos y array de estos tipos. Las palabras reservadas

para definirlos son int, bool y array int /bool.

Antes de definir todos los operadores, empezaremos con el operador //, que usaremos para escribir comentarios. Su sintaxis es un operador por linea, y es un operador prefijo, se escribe antes del comentario a escribir.

Como **operadores infijos** entre enteros y reales (siempre que estén definidas, por ejemplo, no podemos dividir entre 0 en la semántica, aunque sí para la sintaxis), están los operadores infijos +, -, *, /, $\wedge$, $\div$ con su definición de la aritmética matemática usual, siendo los dos últimos el operador potencia y módulo. También se tienen los operadores infijos ==, !=, <=, >=, <, > entre enteros, que generan un valor booleano true o false, dependiendo si el número de la izquierda es igual/distinto/menor o igual/mayor o igual/menor/mayor que el número de la derecha.
Por último, en cuanto a **operadores infijos** entre booleanos, tenemos los operadores == y != (igualdad o desigualdad), y tenemos otros tres operadores booleanos, con las palabras reservadas and y or, que tendrán la función del and y or respectivamente.

Por último, como **operador unario postfijo** tenemos el operador . que usaremos para acceder a los distintos campos de un struct. En la siguiente tabla condensamos toda la información sobre los operadores, donde el operador más prioritario tiene una prioridad de 0, y cuanto mayor sea el número de la tabla, menos prioritario será:

Operador	Preferencia	Asociatividad	Parámetros	Devuelve	Sintaxis
[ ]	0	Izq a Der	Array y un entero posición	Tipo del array	var_array[ent]
( )	0	Izq a Der	Función	-	var_función(parámetros)
.	0	Izq a Der	Struct	Campo del struct	var_struct.campo
!	1	Der a Izq	Booleano	Booleano (negado)	!var_bool
$*$	2	Izq a Der	Enteros y reales	Entero y reales	num $*$ num
/	2	Izq a Der	Enteros y reales	Entero y reales	num / num
$\div$	2	Izq a Der	Enteros y reales	Entero	num $\div$ num
+	3	Izq a Der	Enteros y reales	Entero y reales	num + num
−	3	Izq a Der	Enteros y reales	Entero y reales	num − num
<	4	Izq a Der	Enteros y reales	Booleano (true/false)	num < num
>	4	Izq a Der	Enteros y reales	Booleano (true/false)	num > num
==	5	Izq a Der	Enteros, reales y booleanos	Booleano (true/false)	var == var
! =	5	Izq a Der	Enteros, reales y booleanos	Booleano (true/false)	var ! = var
and	5	Izq a Der	Expresiones booleanas y numéricas	Booleano (true/false)	exp *and* exp
or	5	Izq a Der	Expresiones booleanas y numéricas	Booleano (true/false)	exp *or* exp
=	6	Der a Izq	Variable y valor (u otra variable)	-	var = valor (var)

Destacar que cualquier expresión rodeada de paréntesis es más prioritaria.
Para la **definición de tipos de usuario**, usaremos la palabra reservada typedef y la sintaxis será:

typedef tipo identificador;

IMPORTANTE Es importante destacar que el bloque de typedefs debe ser

el primer bloque de nuestro proyecto y que estos tipos serán usados en structs, no en el programa principal. Algunos ejemplos:

```
typedef int iden_t;
typedef bool iden_t;
typedef array int lista_t;
```

5. Instrucciones

En cuanto a la **instrucción de asignación**, tenemos varias maneras de asignar un valor a una variable, todas se harán con el operador = :

1. En la misma linea de la definición de la variable, con la sintaxis:

   tipo identificador_variable = valor;

2. Asignación de un valor a una variable que ha sido definida previamente, con la sintaxis:

   identificador_variable = valor;

3. Para variables de tipo array, podemos asignar el valor a uno de sus elementos, usando la sintaxis:

   iden_array [entero_positivo]...[entero_positivo] = valor;

4. Se puede asignar a una variable el valor que devuelve una función, metiendo por parámetros de la función operaciones aritméticas, variables y accesos a la posición de un array o a un campo de una variable declarada como struct dentro del parámetro de la llamada. Se tiene la siguiente sintaxis:

   (tipo) iden = iden_función (var_1/val_1,...,var_n/val_n);

Cabe destacar que cuando nos referimos a valor, también nos referimos al valor que tiene una variable, pudiendo tener el identificador de dos variables distintas a ambos lados del operador =, modificando el valor de la variable de la izquierda (y el del parámetro si en la definición de la función se pasa con &)(si es un array se modifica la posición indicada entre [ ]). La sintaxis se ve en este ejemplo:

```
int a=5;          bool b=true;        array bool lista=[true,false];
int n;            bool p;             bool t;
n=50;             p=false;            t=lista[0];
int s=suma(2,3);                      lista[1]=true;
```

En cuanto a la **instrucción condicional**, usamos las palabras reservadas if y else. Para explicar la sintaxis usaremos un ejemplo:

```
if(cond1){
  //Entra aqui si la expresion booleana cond1 evalua a true
}
else{
  //Entra aqui en caso contrario
}
```

En cuanto a **bucles**, tendremos dos tipos: *while* y *for*. En cuanto a su sintaxis:

while (expresión_booleana) bloque

for (int iden = valor; expresión_booleana; expresión_aritmética_iden) bloque

En la siguiente imagen se ve un ejemplo de cada uno, que ejecutan el mismo código, es decir, son equivalentes:

```
int i=0;
array int lista=[0,0];
while(i<2){
  //Entra aqui si la expresion booleana de dentro del parentesis
  //es true
  //Dentro de este bloque ira codigo que modifique
  //la condicion del while, si no formara un bucle infinito
  lista[i]=i;
  i=i+1;
}

for(int j=0;j<0;j=j+1){
  lista[j]=j;
}
```

Para instrucciones de entrada/salida, tenemos las palabras reservadas **read** (lectura) y **print** (escritura), siguiendo la sintaxis:

read iden_var;
print iden_var;

Por último, tenemos la **instrucción return**, en cuya sintaxis se usa la palabra reservada return, que servirá para devolver un valor dentro de una función (solo tipos básicos), como en el ejemplo:

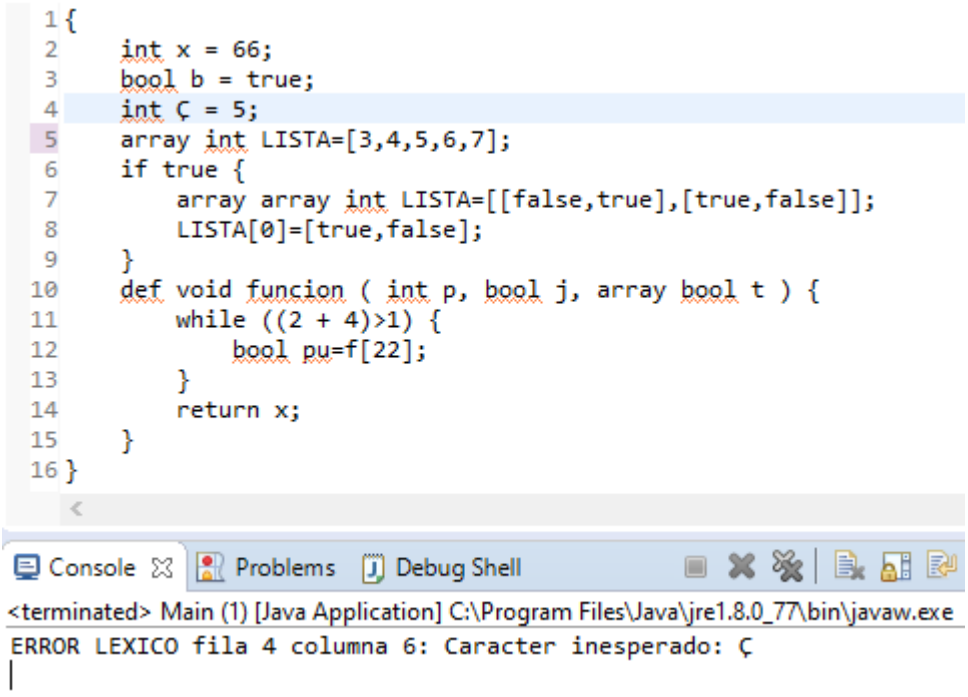
```
def int devuelve_var (int var){
  return var;
}
```

IMPORTANTE Si queremos una función que devuelva un array, debemos hacerlo por parámetro por referencia.

6. Gestión de errores

Los ficheros **ErrorSint.txt**, **ErrorTipado.txt** y **ErrorVinc.txt** tienen ejemplos de recuperación de errores.

Definiremos los errores léxicos y sintácticos. Un error léxico se da si el analizador léxico encuentra un carácter inesperado. Un error sintáctico se produce cuando se escribe código de una forma no admitida por las reglas del lenguaje, ya mencionadas en las secciones anteriores. De momento, nos limitamos a lanzar un mensaje de error diferenciándolos entre ellos (cada tipo de error), como se puede ver en las dos imágenes siguientes:



```
1 {
2     int x = 66;
3     boool b = true;
4     array array int LISTA=[3,4,5,6,7];
5     if true {
6         array array int LISTA=[[false,true],[true,false]];
7         LISTA[0]=[true,false];
8     }
9     def void funcion ( int p, bool j, array bool t ) {
10        while ((2 + 4)>1) {
11            bool pu=f[22];
12        }
13        return x;
14    }
15 }
```

```
Console  Problems  Debug Shell
<terminated> Main (1) [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe
ERROR SINTACTICO fila 3 columna 8: Elemento inesperado "b"
```

Cabe destacar que si hay errores tanto léxicos como sintácticos no se seguirá con la vinculación, tipado ni generación de código.

La gestión de errores en la parte de vinculación y tipado se gestionan mandando mensajes de error sin interrumpir la ejecución, pero sin saltar a la etapa siguiente. Por ejemplo si hay un error en la vinculación no pasará al tipado, y si lo hay en el tipado no generará código. El programa especificará con detalle qué es lo que ha ocurrido, como se ve en este ejemplo, que se puede encontrar en el fichero **ErrorTipado.txt**:

```
{
    {
        typedef int t_s;
    }
    {
        struct t_st{
            int campo1;
            bool campo2;
        }

    }
        {
        def int suma ( int a , int b ) {
            int sum=a+b;
            return sum;
        }
    }
    {

    t_st hola;
    hola.campo1=true;
    int x=2 + 3;
    bool z=8;
    int s=suma(x,z);
    bool aaa=hola.campo2;
    bool y = false;
    y=aaa;
    array(3) int lista;
    x=lista;
    print x;
    }
}
```

Console | Problems | Debug Shell | Search

```
<terminated> GENERA [Java Application] C:\Program Files\Java\jre1.8.0_77\bin\javaw.exe (15 ma
Error en tipado. Incompatibilidad de tipos:
Incompatibilidad de tipos en asignación con Tipo1: int Tipo2: bool
Error en tipado. Incompatibilidad de tipos:
Incompatibilidad de tipos en asignación con Tipo1: bool Tipo2: int
Error en tipado. Incompatibilidad de tipos:
Incompatibilidad de tipos en asignación con Tipo1: int Tipo2: bool
Error en tipado. Incompatibilidad de tipos:
Incompatibilidad de tipos en asignación con Tipo1: int Tipo2: array(int)
Ha habido errores en el tipado y no seguimos con la generacion de codigo
```

7. Ejemplo de programa

```
{
    {
        typedef int num_pedido_t;
    }

    {
        struct pedido_t{
          num_pedido_t nump;
          int coste;
        }
    }

    {
        def void suma_coste (int & p,int s){
          p=p+s;
        }

        def int suma_costes (array int lista, int tam){
          int total=0;
          for(int i=0;i<tam;i=i + 1){
            total=total+lista[i];
          }
          return total;
        }
    }

    {
        array (8) int lista;
        int i=0;
        while(i<8){
          pedido_t p;
          p.nump=i;
          p.coste=i * 10;
          lista[i]=p.coste;
          i=i + 1;
        }
        suma_coste(lista[6],8); //lista[6] tiene coste 60 y pasa a tener 68
        int coste_total=suma_costes(lista,8);   //Sumas todos 0+10+20+30+40+50+68+70 y da 288
        int bonus=5;
        if(coste_total>100){    //Se mete aqui
          coste_total = coste_total - 10 * bonus;
        }
        else{   //Para que se meta aqui reduce el segundo parametro de sumacostes a 5 o menos
           coste_total=coste_total- bonus*10/5;
        }
        print coste_total; //Da 238 en este ejemplo ya que se mete en el if
    }
}
```

En la carpeta este ejemplo se puede encontrar en **Ejemplo1.txt**. Se pueden encontrar otros ejemplos (código con un sentido y funcional) y ficheros de prueba (código funcional pero sin sentido para probar todas las implementaciones). Los ficheros de prueba son códigos que he usado para implementar el lenguaje y depurar errores, y me parecía interesante añadirlos. El fichero .wat que se genera es el **codigo.wat** y el .js a ejecutar es el **main-memory.js**. El fichero **ejecutar.txt** tiene todas las instrucciones para ejecutar todo. En la carpeta también se encuentra el código .wasm ya generado para este ejemplo.

Respecto a los demás ejemplos, el ejemplo **Ejemplo1.txt** es bueno para probar typedefs, structs, acceso a ellos, listas, instrucciones condicionales y distinto tipos de funciones. El **Ejemplo2.txt** es el algoritmo de multiplicación de matrices 3x3, bueno para probar el paso de arrays por parámetros (tanto por valor como referencia), bucles, accesos a arrays y operaciones aritméticas. El **Ejemplo3.txt** es el algoritmo de calculo del numero combinatorio

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \tag{1}$$

donde se puede probar la recursividad de funciones y la llamada de funciones ya definidas anteriormente.

8. Modificaciones respecto a la idea inicial

A continuación enumero los cambios respecto a la idea inicial:

1. Eliminación de punteros (solo esta en el léxico y sintáctico).
2. Return solo de tipos básicos (para devolver arrays hacerlo por parámetro por referencia).
3. No hay listas de structs.
4. No se pueden pasar structs por parámetro.
5. No se puede inicializar arrays multidimensionales directamente asignándole una lista de listas, hay que usar una función auxiliar creada por el usuario.
6. En el paso por parámetro de arrays multidimensionales se debe especificar el tamaño de cada dimensión del array.

El resto está todo implementado, como se puede comprobar en los ejemplos.