ESPECIFICACIÓN DEL LENGUAJE

IVRO®

"Implementado por mentes informáticas, pensado para almas matemáticas"

Rodrigo Cuadro López e Iván Nieto Peralvo

11 de mayo de 2025

Motivación

La idea de este lenguaje surge a raíz de la educación matemática que hemos recibido en el doble grado en paralelo con la informática. Hemos diseñado este lenguaje motivados por la dificultad que puede suponer a las personas con un trasfondo plenamente matemático, introducirse en el mundo de la programación. A nuestro parecer, una sintaxis más cercana a la forma de expresar ideas en lenguaje matemático, facilitaría la adopción de otros conceptos más esenciales, evitando dedicar esfuerzo a interiorizar una sintaxis completamente nueva.

Como es lógico, tomamos inspiración del léxico y la sintaxis matemática pura, pero también de lenguajes funcionales como *Haskell*, que a su vez tienen la misma intención que nosotros. Sin embargo, nuestro enfoque es iterativo, lo que diferencia esta propuesta de otros intentos anteriores.

Como introducción previa a la explicación del lenguaje, detallaremos algunas características generales para ir entendiéndolo mejor:

- Los tipos de variables que se podrán declarar en `IVRO` únicamente serán dos: enteros (`int`) y booleanos (`bool`). Se podrán utilizar también estructuras para almacenarlos, en forma de listas (`array`) de varios tamaños, o como un compendio de datos conjuntos (`tuple`).
- Siguiendo la línea de la similitud con C++, todo programa deberá contener la función principal para ejecutarse, el clásico `main`, sin ningún parámetro de entrada y con valor de devolución entero (0 en caso de terminación correcta). Toda la lógica de los programas deberá estar contenida en esta función o ser llamada desde ella.
- Se informará al usuario de lo errores en la compilación de su código a través de la consola, que indicará la línea donde aparece el error y la causa del mismo.
- Al final del documento se incluye un apéndice con las **limitaciones del lenguaje**, funcionalidad que no ha podido ser implementada por falta de tiempo.

Realmente el lenguaje `IVRO` no está registrado.

1. Objetivos

Este listado refleja los objetivos que nos hemos puesto de cara al lenguaje. No descartamos la posibilidad de modificarlos a lo largo del camino, pero nos servirán como base de la que partir.

Identificadores y Ámbitos de Definición

- Declaración variables y arrays.
- Bloques anidados.
- Funciones (con paso de parámetros por valor y por referencia).
- Parte opcional:
  - Punteros.
  - Registros.
  - Sobrecarga de funciones.

Tipos

- Declaración explícita de tipos.
- Tipos int y bool.
- Operadores infijos (asociatividades y prioridades).
- Tipo array.
- Comprobación de tipos.
- Parte opcional:
  - Definición de tipos de usuario.
  - Tipo tupla (struct).

Conjunto de Instrucciones del Lenguaje

- Asignación.
- Condicionales de 1 y 2 ramas.
- Bucles.
- Expresiones con constantes, identificadores con y sin subíndices, operadores infijos.
- Parte opcional:
  - Expresiones con punteros y acceso a tuplas.
  - Instrucción “case” con salto a cada rama en tiempo constante.
  - Instrucciones de reserva de memoria dinámica.

Gestión de Errores

- Indicación del tipo de error con fila y columna.
- Recuperación de errores en compilación.

2. Especificación del Lenguaje

2.1. Comentarios

La sintaxis de los comentarios de una línea ha sido inspirada en C++ (usando //). Sin embargo, los comentarios multilínea se delimitarán por el carácter #.

```
// Comentario de una linea.
#
Este es un
comentario multilinea.
#
```

2.2. Identificadores y Ámbitos de Definición

Declaración de Variables

La declaración de variables en `IVRO` se realiza de la misma forma que en otros muchos lenguajes de programación, indicando primero el tipo y luego su identificador. Esta forma de proceder será aplicable a todo tipo de variables, desde los tipos simples (`int` y `bool`) hasta los más complejos (`arrays` o `structs`). Si se desea inicializar la variable en el momento de su declaración, se empleará el operador de asignación ":=".

```
// Variable de tipo generico, con su identificador correspondiente.
tipo var0;
// Se inicializa var a valor.
tipo var := valor;
```

Declaración de Arrays

Se escribirá el tipo de los elementos del array a declarar entre corchetes, escribiendo a la derecha del tipo una coma seguida de un entero que indique su tamaño. Este entero es obligatorio, ya que no se contempla la posibilidad de arrays de tamaño variable en la pila. Para inicializar un array, bastará usar el operador de asignación seguido de una lista de elementos (del tipo correspondiente) entre corchetes y separados por comas.

```
// Ejemplo de declaracion de array unidimensional,
[tipo, tam] mi_arr;
# multidimensional (se puede poner cualquier tipo entre los corchetes,
incluyendo arrays), #
[[[tipo, tam1], tam2], tam3] mi_arr;
// y dos inicializados:
[int, 5] mi_arr := [1,2,3,4,5];
[[int,2],2] mi_mat := [[1,2],[3,4]];
```

Declaración de Alias de Tipos

Tomando inspiración del abundante uso de los cambios de notación en matemáticas, se usará la palabra `denote` seguida del tipo a renombrar, la palabra reservada `as` y el alias.

```
denote [[int,2],2] as mat2x2
```

Declaración de Punteros y Memoria Dinámica

En nuestro lenguaje, existirán punteros a variables almacenadas en el heap. El símbolo que establece que una variable es de tipo 'puntero a" es el `@` (análogo a `*` en C++).

Por otro lado, de la misma manera que con las constantes, para la utilización de memoria dinámica la sintaxis será análoga a la de C++, sustituyendo el `new` por `newMem`.

```
int @ puntero_a_int := newMem int; // Se reserva memoria.
@ puntero_a_int := 3; // Se modifica el contenido en el heap.
write(@ puntero_a_int); // Se imprime el valor de esta expresion (3).

// Tambien se pueden declarar punteros de otras formas:
[int @, 40] mi_arr;
```

Declaración de Funciones

En línea con la idea de darle el mayor *feeling* matemático al código[1] se toma la notación habitual para definir funciones ($f : A \longrightarrow B$), adaptándola al mundo informático. Primero se escribe el nombre de la función, seguido de los parámetros de entrada, los caracteres "`->`" y finalmente los parámetros de salida (en caso de no recibir o devolver nada, se usará `empty`, por el conjunto vacío). Para distinguir entre paso de parámetros por valor y por referencia (sólo los tipos simples), nos inspiraremos también en C++, usando el símbolo `?` para denotar aquellos parámetros que se pasen por referencia.

Como detalle destacable, en `IVRO` se permite la sobrecarga de funciones, lo que puede ser muy útil para el usuario. Sin embargo, no se permite el paso de punteros como parámetro, ya que se entiende que las referencias cumplen el mismo cometido.

```
// Funcion que devuelve el maximo de un array.
// En el paso de arrays por valor es necesario indicar el tamano.
maximo: ([int, 10] elems, int tam) -> int{...}

// Funcion que recibe dos punteros a entero e intercambia sus contenidos.
swap_int: (int ? e1, int ? e2) -> empty {...}
```

2.3. Tipos

Se dispondrá de enteros (`int`) y booleanos (`bool`), este último pudiendo ser "`True`" o "`False`". Los `arrays` han quedado descritos en su totalidad en la sección anterior. Falta describir la declaración y el uso de structs, así como los operadores infijos de los que dispondrá el lenguaje.

Structs

En `IVRO`, los structs serán conocidos como `tuple`. Para declararlos, se comenzará por la palabra `tuple`, se indicará entre paréntesis los tipos de los elementos, seguidos (entre llaves) de los identificadores de cada campo separados por comas (pudiendo asignarles un valor inicial). Por último, el identificador de la estructura.

En cuanto al acceso a sus elementos, se hará de forma similar a los arrays, pero sustituyendo los corchetes por un punto seguido del identificador del campo.

[1] para nuestros amigos al otro lado del parque

```
// Ejemplo de un struct con tres parametros (var2 sin inicializar).
tuple (bool, int, int) {
    var1 := True,
    var2,
    var3 := -33
} mi_tupla;

// Acceso a los elementos 1 y 3 de la tupla.
mi_tupla tupl; // Los valores se inicializan segun la declaracion.
bool mi_bool := tupl.var1;
write(tupl.var3); // Esta inicializado al valor -33
int mi_int := tupl.var3;
```

Operadores Infijos

Se emplearán los usuales en matemáticas, salvo por algunas excepciones. Estas son la potenciación, donde se usará '^' (como en LaTeX); el "and" lógico, que se representará con '&'; el "or", para el que se empleará '|'; y el "not", denotado por '!'. Se añadirá también el operador '%' para el módulo. Recalcar nuevamente que el símbolo de asignación será "`:=`" en lugar de "`=`", siguiendo la notación matemática de las definiciones.

Para acceder a una posición de un array se empleará el operador "`[i]`" tras el identificador del array, donde `i` es la posición que se consultará. Si el array tiene tamaño n, las posiciones se identificarán con los números del 0 al $n-1$. Para arrays de mayor tamaño, se accederá a sus respectivas posiciones con sucesivos accesos a cada submatriz

OPERADOR	PRIO	TIPO	ASOCIATIVIDAD
:=	0	BIN	NO
|	1	BIN	IZQ
&	2	BIN	IZQ
!	3	UN	NO
=, !=	4	BIN	NO
<=, >=, <, >	5	BIN	NO
+, -	6	BIN	IZQ
*, %, /	7	BIN	IZQ
^	8	BIN	IZQ
@	9	UN	IZQ
[]	10	BIN	IZQ
.	11	BIN	DCHA

Cuadro 1: Tabla de operadores con prioridad, tipo y asociatividad

```
// Ejemplo de la potencia al cubo de 2.
2^3
// Ejemplo sencillo de una posible logica.
true & (False | !B)
// Dada la matrix 2x2 [int, 2], 2] mi_mat, acceso a la posicion (1,2).
mi_mat[0][1]
```

2.4. Instrucciones del Lenguaje

Condicionales

Para las instrucciones condicionales se tomará la sintaxis de C++, ya que su forma de plantearlos es ligera y eficaz. Y lo que es más importante: comprensible para un matemático sin conocimientos previos.

```
if (bool condicion1){
// Codigo a realizar si condicion1 es True.
}
else if (bool condicion2){
    // Codigo a realizar si condicion1 es False y condicion2 es True.
}
else{
    // Codigo a realizar en otro caso.
}
```

Bucle `while`

La creación de este bucle es sencilla, pues únicamente está formado por la palabra `while` seguida de la condición a cumplirse entre "()". Al igual que en C++, primero se comprueba la condición. En caso de cumplirse, se ejecuta el cuerpo del bucle y nuevamente se vuelve a la condición inicial. En caso contrario, termina la ejecución del bucle.

```
while (bool condicion) {
    // Cuerpo del bucle.
}
```

Bucle `for`

La construcción de este bucle es un poco más elaborada que la del `while`, pero su presentación es casi idéntica a la de C++. Comienza con la palabra `for`, seguida de tres sentencias entre paréntesis separadas por "`;`":

- La primera sección contendrá una variable de tipo `int`, que debe ser declarada en el momento. El uso esperado es que esta variable se use para contabilizar el número de vueltas, pero podría ser otro.
- La segunda sección debe contener una expresión booleana, que causará la salida del bucle cuando sea False (si llega a serlo). Típicamente estará relacionada con el contador de la primera sentencia.
- La tercera sección contiene una expresión que podría ser cualquiera, pero pensada para ser una expresión aritmética que afecte al contador haciendo que, por ejemplo, aumente o disminuya su valor.

```
for (int i := 1; bool condicion; i := i + 1) {
    // Cuerpo del bucle.
}
```

Bucle `repeat`

Este bucle tiene la misma funcionalidad que el `while`, con la particularidad de que ejecuta su cuerpo antes de comprobar la condición. Esto implica que sea cierta o falsa la condición, su cuerpo se siempre se ejecutará al menos una vez. Tras esto, su funcionamiento pasa a ser el mismo que `while`. Esta funcionalidad puede ser muy útil si se espera que una variable tome un valor permitido en el bucle para una acción posterior, ya que se evita tener que inicializarla previamente. Además, permite no distinguir casos en ciertas ocasiones.

```
repeat{
    // Cuerpo del bucle.
} until (bool condicion);
```

Funciones de Entrada y Salida

La lectura y escritura en `IVRO` se manejará a través de las funciones de entrada `read()` y salida `write()`. Estas funciones accederán al "buffer" de entrada o salida y devolverán el primer entero (o booleano) que haya. En caso de encontrarse una cadena inválida para el tipo, fallarán.

Cabe destacar el uso de la sobrecarga de funciones para simplificar la entrada y salida de datos, unificando las funciones bajo un mismo nombre.

```
// La funcion read() lee un valor y se lo asigna a la variable x.
int x;
read(x);

// Mostramos por pantalla el valor de la variable x con write().
write(x);
```

Switch Statement

Otro gran conocido en los lenguajes de programación. Su estructura será muy similar a su versión en C++, pero con cambios en la notación para acercarlo más a la jerga matemática, con inspiración en las demostraciones por casos donde cada valor posible de una variable puede implicar cosas totalmente distintas. Se incluye la posibilidad de añadir un caso general opcional, destinado a tratar las instancias de la ejecución en las que ninguna de los casos anteriores sea el adecuado.

En cuanto a la sintaxis, comienza con la palabra `incase` seguida por la variable a estudiar por casos. A continuación y entre corchetes se encontrarán los casos, donde cada uno de ellos tiene la siguiente estructura: La palabra `equals` junto con el caso particular a tratar, seguido de los caracteres "`=>`" (inspirado en la implicación matemática) y el código a ejecutar en ese caso entre corchetes. Para el caso genérico, se escribirá "`otherwise =>`".

Durante la ejecución de los programas, se salta al caso adecuado en tiempo constante.

```
incase x {
    equals op1 => {
        // Codigo a realizar.
    }
    equals op2 => {
```

```
        // Codigo a realizar.
    }
    otherwise => {
        // Codigo a realizar.
    }
}
```

Constantes

A la hora de declarar una constante en `IVRO`, se procederá nuevamente de manera similar a C++, con la particularidad de cambiar la palabra `const` por la palabra `cte`, haciendo alusión a la abreviatura que se le da en matemáticas.

```
// Esta variable no podra modificar su valor en el programa.
cte int CONSTANTE := 10;
```

3. Limitaciones del Lenguaje

A lo largo del desarrollo del proyecto, nos hemos encontrado con pequeños detalles que, por falta de tiempo, no hemos podido pulir del todo. Nos gustaría resaltar un par de casos que nos parecen los más relevantes.

El primero es el acceso a arrays almacenados en el heap. Es un problema relacionado con la gramática, ya que pese a que la declaración es correcta, al intentar acceder a una posición con la instrucción:

```
[int, 10] @ arr2 := newMem [int, 10];
write((@ arr2)[4]); // Instruccion problematica.
```

la gramática entiende incorrectamente el acceso como expresión. Estamos seguros de que con más tiempo sería posible hacer la modificación adecuada para solucionarlo.

Otro sería el hecho de que en el paso de arrays a funciones por valor no hemos conseguido implementar la sobrecarga. Esto se debe a que, en caso de usarla, el compilador se ve forzado a comparar los tipos de los parámetros de las funciones, pero debido a que el tipado aún no se ha hecho, el tamaño de los arrays se desconoce y por tanto su tipo no se puede conseguir.

No hemos podido identificar más errores que nos hayan parecido destacables a pesar de todo el testing que hemos llevado a cabo, pero no descartamos la posibilidad de que exista alguno más.

Con todo, estamos satisfechos con el resultado, ya que hemos conseguido implementar lo que nos parece un amplio catálogo de herramientas en nuestro lenguaje (gran parte de lo que nos propusimos en un principio). Esta experiencia nos ha ayudado a comprender la gran cantidad de sutilezas que hay detrás de un lenguaje de programación que pasan tan desapercibidas en el día a día.

4. Ejemplos de uso

En esta sección se muestran ejemplos paradigmáticos de programas en los que se emplean las diversas características del lenguaje, a través de ejemplos lo más ilustrativos posible.

Ejemplo 1:

Este programa es una prueba de la funcionalidad de los bucles presentes en el lenguaje.

```
main: empty -> int{
        int hola := 2;
        [int, 3] arr := [1,2,3];

        for(int i := 0; i < 3; i := i + 1){
                write(arr[i]);
                hola := hola + i;
        }
        write(hola);

        int j := 0;
        while( j < 3){
                write(arr[j]);
                hola := hola * (j + 1);
                j := j + 1;
        }
        write(hola);

        j := 0;
        repeat{
                write(arr[j]);
                hola := hola - j;
                j := j + 1;
        } until (j = 3);
        write(hola);

        return 0;
}
```

Ejemplo 2:

Este programa comprueba el uso de constantes, alias y condicionales.

```
cte int TAM := 10;
denote [int, TAM] as array;

main: empty -> int {
        array arr;

        for(int i := 0; i < TAM; i := i + 1){
                arr[i] := i + 1;
        }

        bool b1 := True;
        bool b2 := False;

        if(arr[TAM - 1] < 10){
                write(TAM / 3);
        }
        else if(arr[TAM - 1] = 10 & !b2){
                if((b1 | b2) & !False){
                        write(777);
                }
                write(TAM / 5);
                write(TAM % 5);
        }
        else{
                write(1);
        }

        return 0;
}
```

Ejemplo 3:

En este caso, se emplean arrays multidimensionales y alias, además de la instrucción incase-equals.

```
denote int as entero;

main: empty -> int{
        entero n := 4;
        write(n);

        [entero, 5] arr := [1,2,3,4,5];
        write(arr[n]);


        [[entero,2],3] mat := [[1,2],[3,4],[5,6]];

        write(mat[0][0]);
        write(mat[0][1]);
        write(mat[1][0]);
        write(mat[1][1]);
        write(mat[2][0]);
        write(mat[2][1]);


        for(int i := 0; i < 3; i := i + 1){
                for(int j := 0; j < 2; j := j + 1){
                        write(mat[i][j]);
                }
        }

        incase n {
                equals 2 =>{
                        write(n + 1);
                }
                equals 3 =>{
                        write(n + 17);
                }
                otherwise => {
                        write(0);
                }
        }


        return 0;
}
```

Ejemplo 4:

El siguiente código muestra la sobre carga de funciones y el paso de parámetros por referencia.

```
suma1: int a, int b -> int {
        return a + b;
}

suma: int a, int b -> int {
        int result := a + b;
        return result ;
}

suma: int a, int b, int ? result -> empty {
        result := a + b;
}

sumaElems: [int, 10] arr, int n -> int {
        int result := 0;
        for(int i := 0; i < n; i := i + 1){
                result := result + arr[i];
        }
        return result;
}

main: empty -> int{

        int num := suma1(53, 14);
        write(num);

        num := suma(num, -10);
        write(num);


        int result := 3;
        suma(10, 52, result);
        write(result);

        [int, 10] arr := [1,2,3,4,5,6,7,8,9,10];
        write(sumaElems(arr, 10));

        return 0;
}
```

Ejemplo 5:

Ahora, se prueban las funciones de I/O y la memoria dínamica (y por tanto el uso de punteros).

```
main: empty -> int{
        // Pruebas de I/O
        int num;
        bool b;

        read(num);
        write(num);

        read(b);
        write(b);

        //Pruebas de Punteros
        int @ hola := newMem int;
        @ hola := 3;
        write(@ hola);
        delete hola;

        [int @, 10] arr;

        for(int i := 0; i < 10; i := i + 1){
                arr[i] := newMem int;
                @arr[i] := i*3;
                write(@arr[i]);
        }

        for(int i := 0; i < 10; i := i + 1){
                delete arr[i];
        }

    // Los arrays pueden ser declarados en el heap, pero el
    // acceso no funciona.
        [int, 10] @ arr2 := newMem [int, 10];

        return 0;
}
```

Ejemplo 6:

En este, se muestra la funcionalidad de las tuplas, incluyendo un array como uno de los campos.

```
tuple (int, bool, [int,5]){
        v1 := -17,
        v2 := True,
        v3
} mi_tupla;


main: empty -> int{
        mi_tupla t;

        write(t.v1);
        write(t.v2);

        t.v3 := [0,2,4,6,8];

        for(int i := 0; i < 5; i := i + 2){
                write(t.v3[i]);
        }

        return 0;
}
```

Ejemplo 7:

En el último, se prueba a calcular de forma recursiva el elemento n-ésimo de la sucesión de Fibonacci, para asegurarnos de que la implementación de las llamadas a funciones es correcta.

```
fib: int n -> int{
        int res;
        if(n < 3){
                res := 1;
        }
        else{
                res := fib(n - 1) + fib(n - 2);
        }
        return res;
}

main: empty -> int{
        int n;
        read(n);
        int result := fib(n);
        write(result);

        return 0;
}
```