UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

PROCESADORES DE LENGUAJES

DIAL

Alejandro Parreño
Diego Ostos

Doble Grado en Ingeniería Informática y Matemáticas

Febrero 2025

Índice

1. Introducción

En este documento se incluyen las características y especificación de la sintaxis del lenguaje de programación que pretendemos construir, junto a ejemplos que consideramos oportunos para familiarizarse con el lenguaje y su sintaxis.

2. Especificación del lenguaje

En este lenguaje, las instrucciones terminarán por ; . Entre los puntos fuertes de este lenguaje destacamos la alta compatibilidad entre reales y enteros, el uso flexible de constantes arrays y records, tipos enumerados, alias de tipos, instrucción condicional ternaria al estilo C++, una instrucción switch con un alto grado de libertad sintáctica, accesos parciales a arrays, y muchas más cosas que se detallan en este documento.

2.1. Comentarios

Contaremos con dos tipos de comentarios. Los comentarios de una sola línea, los cuales irán precedidos por el carácter #, mientras que los comentarios en bloque comenzarán con tres comillas dobles """ y terminarán de la misma manera.

```
# Esto es un comentario de una linea
```

```
"""
Esto es un comentario
de varias lineas
"""
```

2.2. Identificadores y ámbitos de definición

2.2.1. Declaración de variables simples

Consistirá en indicar el tipo de la variable seguido del identificador.

```
type variable;
```

Declarar e inicializar una variable de forma simultánea se hará de forma similar al caso anterior, seguido el identificador de la variable por el carácter = y a continuación, el valor con el que se pretende inicializar.

```
type variable = valor;
```

2.2.2. Declaración de arrays

Declarar arrays sin inicializar se hará indicando la palabra reservada array seguida del tipo de los elementos del array, seguido de corchetes [y], donde deberá aparecer un número que será el tamaño, seguido del identificador del array. Cabe destacar que se permiten arrays de arrays si el tipo que precede a los corchetes vuelve a ser array y en ese caso no hay que volver a indicar la palabra clave array.

```
array type[5] array1;
```

Para declarar e inicializar un array, se puede hacer declarando el array del mismo modo que antes, pero a continuación del identificador seguirá un =, y tras este, entre corchetes [y], y separados por comas, los elementos del array en caso de utilizar una expresión constante.

```
array int[2][4] array1 = [[1,2,3,4], [3,4,5,6]];
array int[2][4] array2 = array1;
```

A su vez, se permite el uso de arrays parciales dentro del lenguaje pudiendose así usar solamente parte de los mismos

```
array type[2][2] matriz = [[1,2],[3,4]];
array type[2] fila1 = matriz[0];
```

2.2.3. Declaración de funciones

Las funciones serán declaradas de la siguiente manera, primero la palabra reservada func, seguido de esta el tipo de valor de retorno de la función. En el caso de una función sin retorno, la identificamos con none. Después, el identificador de la función y entre paréntesis los parámetros de entrada de la función. Los parámetros se expresan por medio del tipo del parámetro y un identificador separados por comas. Admitimos paso de parámetros por valor y referencia, siendo el primer caso el que se entiende por defecto, y para explicitar que el paso es por referencia, el tipo del parámetro deberá ir precedido por el símbolo &. El cuerpo de la función vendrá a continuación de los parámetros y limitado por llaves, { y }. Será necesario añadir una instrucción de return para devolver un valor del tipo de la función salvo que el tipo sea none, donde admitiremos

tanto un return que no devuelve nada, así como no poner nada directamente.

```
    # Funcion con valor de retorno
    func type name(type var1, type var2,...) {
        type res = ...;
        return res;
    }
    # Funcion sin valor de retorno
    func none name(type var1, type var2,...) {
        return;
    }

    # Funcion con paso por referencia
    func none name(type &var1, type &var2,...) {

    }
```

2.2.4. Declaración de punteros

Para declarar un puntero se declarará el tipo del valor al que se apunta seguido de @, y del identificador del puntero. En el caso de inicializar al mismo tiempo el puntero, se hará de forma idéntica a variables simples y arrays. Además, para reservar memoria dinámica en la inicialización, se utilizará la palabra reservada new seguida del tipo. Para liberar la memoria dinámica, utilizaremos la palabra reservada del, seguida del identificador del puntero, aunque realmente no hacemos nada en generación de código pero no hemos eliminado la instrucción. Por último, para representar el valor de un puntero nulo, se utiliza la palabra clave null.

```
    # Declaracion de puntero sin inicializacion
    type@ ptr;
    # Declaracion de puntero con inicializacion
    type@ ptr = null;
    #Reserva de memoria dinamica
    type@ ptr = new type;
    # Liberar memoria del puntero
    del ptr;
```

Para acceder al valor que apuntan los punteros, usamos una vez más @, seguido del identificador del puntero.

```
    # Ejemplo de acceso a valor de puntero
    type@ ptr = new type;
```

```
3     @ptr = valor;
```

2.2.5. Declaración de registros

La definición de registros se hace explícita por medio de la palabra reservada record seguida del nombre del registro. Entre llaves { }, irán declarados los atributos del registro, cada uno por medio de su tipo seguido de un identificador. Para acceder a cada uno de los atributos del registro, se usa el '.', es decir, id_struct.var. Para inicializar un struct, deben aparecer los valores que se pretende dar a sus campos entre llaves { y }. Admitimos registros vacíos y tenemos especial cuidado con los bucles infinitos que se pueden dar al declarar un registro.

```
1     # Definicion de un registro
2     record Registro {
3         type var1;
4         type var2;
5     }
```

```
1     # Declaracion de un registro
2     Registro reg;
3     # Acceso a las variables de un registro
4     reg.var1;
5     # Inicializacion de registro
6     reg = {valor1, valor2};
```

2.2.6. Módulos

Los módulos son declarados en un fichero aparte donde siempre comenzará con la palabra reservada module, seguida del nombre del módulo, y cada una de las funciones, variables y registros que estén definidos dentro del módulo y que se pretendan utilizar fuera irán precedidas por la palabra reservada export en su declaración. Los programas que importen el módulo solo tendrán acceso a aquello que es exportado.

```
1     module miModule;
2     type var = valor1;
3     export type var = valor2;
4     export func<type> name(type var1, ...){
5         ...
6     }
```

2.2.7. Bloques anidados

Nuestro lenguaje admitirá bloques anidados en las sentencias condicionales if/elif/else así como con los bloques while, for y repeat. Pero no admitiremos bloques anidados para el condicional reducido. La sintaxis de estos bloques anidados será el nombre del bloque exterior seguido de { y terminando con }. En el interior de estos corchetes, se incluirá el bloque que está anidado.

```
1   if(cond1){
2       while(cond2){
3           # Bloque de codigo del anidado
4       }
5       # Bloque de codigo del if
6   }
7   else(cond3){
8       if(cond4){
9           # Bloque de codigo del anidado
10      }
11      # Bloque de codigo del else
12  }
```

2.2.8. Cláusulas de importación

Las cláusulas de importación consistirán en la palabra reservada import seguida del nombre del módulo a importar. Es importante que el módulo se encuentre en el mismo directorio que el programa. Se debe tener en cuenta que haciendo el binding de forma cuidadosa, evitamos que se pueda acceder a aquello que no es exportado, si bien se puede utilizar indirectamente.

```
1   import miModule;
```

2.3. Tipos

2.3.1. Tipos básicos (enteros, reales, booleanos y enumerados)

- El tipo entero se declara por medio de la palabra reservada int.

```
1   int numero = 42;
```

- El tipo real se declara por medio de la palabra reservada real Además, haremos uso del "." para separar la parte entera de la parte decimal. Es importante destacar una vez más que en este lenguaje en todo aque

sitio donde se espere un real podrá ir un entero gracias a la flexible compatibilidad de tipos, incluso cuando los tipos son arrays o registros.

```
    real pi = 3.14;
```

- El tipo booleano se declara por medio de la palabra reservada bool. Los valores booleanos se representan por medio de las palabras reservadas true y false.

```
    bool esVerdadero = true;
    bool esFalso = false;
```

- Los tipos enumerados se pueden declarar de forma similar a los registros, haciendo uso de la palabra reservada enum seguida del nombre del tipo enumerado, y a continuación, entre llaves { y }, deben aparecer los valores posibles para el tipo, separados por comas. Los nombres de los tipos enumerados así como los nombres de los valores seguirán las mismas reglas que los identificadores.

```
    enum Nombre {
        valor1,
        valor2,
        valor3
    };
```

2.4. Compatibilidad de tipos

Además de permitir la declaración y uso de variables de tipo int y real dentro del lenguaje se permite la compatibilidad entre los mismos pudiéndose:

- Asignar un entero a un real

```
    real x = 4;
```

- Devolver un entero en una función de tipo real

```
    func real funcion(int x){
        return x;
    }
```

- Declarar un array de tipo real con enteros y reales

```
    array real[2] arr = [1,3.2];
```

Y cualquier tipo de asignación que se nos ocurra, además de ser compatibles en las expresiones de todo tipo. En los ejemplos de código se pueden ver cosas mucho más complejas con arrays de reales, registros con reales y enteros, étc.

2.4.1. Tipo Array

Como ya se ha indicado anteriormente, un tipo array vendrá dado por un tipo seguido de los corchetes [y].

2.4.2. Alias de tipo

La declaración de alias de tipo se realizará utilizando la palabra reservada using seguida del alias a la izquierda de un =, y a la derecha quedará el tipo para el cual se define el alias.

```
1     using PunteroEntero = int@;
```

2.4.3. Operadores

Los operadores aritméticos de nuestro lenguaje siguen la misma sintaxis que C++, del mismo modo que los operadores booleanos. Incluimos además, los mismos operadores de asignación, que permiten realizar una operación aritmética y una asignación a la vez. Los operadores de acceso a atributos de registros, puntero, así como el operador condicional ternario son especificados en el resto del documento. A continuación detallamos el tipo, asociatividad y prioridad de todos ellos.

Operador	Tipo	Prioridad	Asociatividad
`.`	Binario Infijo	0	Izquierda
`@`	Unario Prefijo	1	Asociativo
`* / %`	Binario Infijo	2	Izquierda
`+ -`	Binario Infijo	3	Izquierda
`<= >= < >`	Binario Infijo	4	Izquierda
`== !=`	Binario Infijo	5	Izquierda
`!`	Unario Prefijo	6	Asociativo
`&&`	Binario Infijo	7	Izquierda
`||`	Binario Infijo	8	Izquierda
`? :`	Ternario Infijo	9	Derecha
`= += -= *= /= %=`	Binario Infijo	10	No asociativo

Cuadro 1: Tabla de operadores en el lenguaje

2.5. Instrucciones del lenguaje

2.5.1. Asignación

La asignación se hará indicando el identificador de la variable a la izquierda del `=` y a la derecha el valor que se pretende asignar. Incluimos también como en C++ asignaciones que son a la vez expresiones aritméticas.

```
1    x = 5;
2    y *= 3;
```

2.5.2. Acceso a arrays

Para acceder a los elementos de una array se pondrá el identificador del array seguido de corchetes `[` y `]`, entre los cuales se indicará la posición a acceder. Comenzamos a numerar las posiciones del array por 0. Se permiten accesos parciales a arrays.

```
1    arrayEjemplo[1] = 5;
```

2.5.3. Condicionales

En nuestro lenguaje se admiten instrucciones condicionales que evalúan expresiones booleanas para determinar cuál bloque de código ejecutar a continuación. Dependiendo del tipo de condicional, se pueden distinguir casos:

- **Una rama.** Se utiliza la palabra clave if seguida de una expresión booleana entre paréntesis. Si la condición se evalúa como verdadera, se ejecuta el bloque de código delimitado por llaves { }. Si es false, se ignora el bloque y se continúa con la ejecución del programa tras el if.

```
    if (condicion1) {
        #condicion1 evalue a true
        bloque1;
    }
```

- **Dos ramas.** Se añade una cláusula condicional else para definir un bloque alternativo. Si la condición del if resulta verdadera, se ejecuta el primer bloque; si es falsa, se ejecuta el bloque definido en el else.

```
    if (condicion1) {
        #condicion1 evalue a true
        bloque1;
    }
    else {
        #condicion1 evalue a false.
        bloque2;
    }
```

- **Múltiples ramas.** Se pueden encadenar varias condiciones utilizando elif. Aquí, el programa evalúa en orden cada condición. Si la primera condición del if es verdadera, se ejecuta su bloque y se omiten las demás. Si es falsa, se evalúa la condición del primer elif. Si esta resulta verdadera, se ejecuta su bloque y se ignoran las siguientes cláusulas. Finalmente, si ninguna de las condiciones es verdadera, se puede incluir o no un bloque else que se ejecutará en tal caso.

```
    if (condicion1) {
        #condicion1 evalue a true.
        bloque1;
    }
    elif (condicion2) {
        #condicion1 evalue a false.
        #condicion2 evalue a true.
        bloque2;
    }
    else {
        #condicion1 y condicion2 evaluen a false.
        bloque3;
```

```
13      }
```

- **Condicional reducido** A su vez, vamos a incluir en nuestro lenguaje también el formato de condicional reducido y su funcionamiento va a ser igual que en C++:

```
1 a = b ? c : d;
```

Lo que es equivalente a un condicional de dos ramas:

```
1 if (b) {
2     a = c;
3 }
4 else {
5     a = d;
6 }
```

Del mismo modo, este condicional reducido se puede extender a múltiples ramas:

```
1 a = b ? c : d ? e : g;
```

2.5.4. Bucles

Tendremos tres tipos de bucles en nuestro lenguaje.

- **While**
  Estos bucles se crearán utilizando la palabra reservada while seguida entre paréntesis de la condición booleana a evaluar en cada iteración y entre llaves { }, el bloque de código a ejecutar cada iteración. El funcionamiento es el mismo que en cualquier otro lenguaje de programación.

```
1     while (condicion) {
2         # Bloque de codigo
3     }
```

- **For**
  Los bucles for se crearán de forma similar a C++. Para ello, se utilizará la palabra reservada for seguida entre paréntesis (y), de la inicialización de un iterador, la condición a evaluar cada iteración y una operación aritmética, separadas por ';', de la misma forma que en C++.

- La inicialización consistirá en declarar e inicializar una variable de tipo entero.
- La condición será una expresión booleana que será evaluada antes de cada iteración, y en caso de verificarse se realizará la iteración, y en caso contrario acabará el bucle.
- La operación aritmética se realizará al final de cada iteración y antes de la evaluación de la condición.

```
1    for (int i = 0; i < 10; i += 1) {
2        print(i);
3    }
```

- **Repeat**
  Este tipo de bucle se comportará de forma similar al do-while de C++. Se construye exactamente de la misma manera que el bucle while, sustituyendo la palabra reservada while por repeat, que también será reservada. La diferencia con el bucle while es que siempre se realizará una primera iteración, si bien luego el comportamiento es exactamente el mismo.

```
1    repeat (condicion){
2        # Bloque de codigo
3    }
```

- **Control de bucles**
  Para el control del flujo en bucles utilizaremos las siguientes palabras clave:
  - break: salir del bucle de forma inmediata.

```
1    for(int i = 0; i < 100; i += 1){
2        #codigo1
3        if (condicion) {
4            #si condicion, salimos del bucle
5            break;
6        }
7        #codigo2
8    }
```

  - continue: saltar a la siguiente iteración del bucle sin terminar la actual.

```
    for(int i = 0; i < 100; i += 1){
        #codigo1
        if (condicion) {
            #si condicion, saltamos codigo2
            continue;
        }
        #codigo2
    }
```

2.5.5. Instrucción case

Tendremos una instrucción case similar al switch de C++ para saltar a diferentes ramas en función del valor de una expresión a evaluar. Esta instrucción se construirá por medio de la palabra reservada case seguida entre paréntesis (y), de la expresión a evaluar, pudiendo ser esta expresión de tipo int o enum. A continuación, entre llaves, { y }, aparecerán cada uno de los posibles bloques de códigos a ejecutar. Cada bloque consistirá en la palabra reservada when, seguida del valor constante con el que se debe comparar la expresión evaluada, y seguido de dos puntos :, y el código a ejecutar. De forma similar a C++, una vez coincida el valor con algún case, los bloques de case por debajo también se ejecutarán, por lo que para evitar esto, permitimos el uso de la palabra reservada break para salir de la instrucción. Por último, el bloque de código que se ejecutará por defecto sin comprobaciones, se indicará con la palabra reservada default al igual que en C++. Cabe destacar el default podrá aparecer en cualquier sitio o no aparecer.

```
    case (expresion) {
        when valor1:
                #codigo1
        when valor2:
                #codigo2
        default:
                #codigoDefault
    }
```

2.5.6. Entrada y salida

Contaremos con dos operaciones de entrada y salida, read y print que se utilizarán para leer y escribir por pantalla respectivamente. La primera será utilizada como función sin argumentos que devuelve lo que lee por pantalla, y la segunda no devolverá nada y recibirá como argumento lo que se pretende

escribir. Print estará sobrecargada para los tipos básicos int, real y booleano. Mientras que read distinguirá con readf y readi para reales y enteros que son los únicos admitidos.

```
print(5);
real x = read();
print(x);
```

2.5.7. Reserva y liberación de memoria dinámica

Tal y como se ha explicado en la declaración de punteros, las instrucciones de reserva y liberación son respectivamente new y del.

2.5.8. Constantes

Las constantes que consideramos son los números enteros, reales, los valores booleanos true y false, y los valores posibles para los tipos enumerados definidos por el usuario. Además de arrays y registros. Como observación las constantes de arrays y registros podrán estar formadas por cualquier expresión que encaje con el tipo como designadores.

2.5.9. Identificadores

Los identificadores podrán incluir letras mayúsculas y minúsculas, dígitos y el carácter "_", con la condición de que el primer carácter no podrá ser un dígito.

```
type _var12; #Identificador valido
type 1myvar_; #Identificador no valido
```

2.5.10. Llamadas a función

Las funciones pueden llamarse mediante su identificador seguido entre paréntesis (y), de los argumentos de la función separados por comas ",". Las llamadas a función podrán devolver arrays y registros, de modo que pueden ser incluso utilizadas como designadores para acceder a sus campos. Además se podrán devolver constantes arrays y registros.

```
type var = name_func(arg1, ..., argN);
```

3. Gestión de errores

Se detectarán los errores durante la compilación, notificando al usuario la fila y columna donde se ubican los errores encontrados, así como su tipo. Se intentará proseguir la compilación con el objetivo de detectar el máximo número de errores posibles y notificarlos.