Entrega 1: Especificación de la Sintaxis del Lenguaje de Programación

Enrique Rey Gisbert y Juan Carlos Llamas Núñez

26 de febrero de 2022

1. Enfoque del Lenguaje de Programación

El objetivo de este documento es presentar la sintaxis del lenguaje de programación que los alumnos Enrique Rey Gisbert y Juan Carlos Llamas Núñez vamos a construir durante el resto del curso. Será un lenguaje imperativo guiado por instrucciones secuenciales similares a las vistas en lenguajes como C++ o Java. Sin embargo, debido a la alta complejidad que puede tomar realizar un compilador para un lenguaje de estas características, nos reduciremos a un subconjunto de instrucciones (que serán suficientes para que el lenguaje sea Turing Completo) y bloques de código con una sintaxis prefijada, además de introducir ciertas modificaciones en algunas instrucciones típicas que simplificarán el proceso de diseño e implementación.

En primer lugar, comentaremos la estructura general que tendrá un programa escrito en este lenguaje de programación, así como las estructuras más amplias que englobarán el resto del código. En segundo lugar, expondremos las unidades léxicas que necesitaremos junto con la sintaxis de las instrucciones que tendrá nuestro lenguaje de programación. Por último, daremos múltiples ejemplos de programas sencillos escritos en el lenguaje a desarrollar, que utilizarán todas las instrucciones que estarán disponibles. Más adelante (en futuras entregas) especificaremos detalladamente el léxico y la sintaxis del lenguaje, y procederemos a exponer la implementación del compilador que vamos a desarrollar.

2. Bloques, Estructuras y Funciones

Un programa en nuestro lenguaje estará compuesto por la declaración (y posible inicialización) de variables globales, y a continuación por la definición e implementación de una serie de funciones. Aunque la sintaxis concreta de las funciones y de las declaraciones (con y sin inicialización) se explicará posteriormente en este documento, cabe destacar que una de estas funciones debe tener *main* como identificador, devolver un número entero y no recibir ningún parámetro. Esta función será por la que comience el hilo de ejecución del programa. En principio no vamos a dar la posibilidad de realizar declaración de tipos propios ni de importación de módulos, aunque no descartamos introducirlos en un futuro. En este caso dichas instrucciones se situarían junto a las declaraciones de variables globales, es decir fuera de los bloques de código de las funciones.

El cuerpo de las funciones estará compuesto por un bloque de código, formado por una secuencia de instrucciones. Algunas de estas instrucciones pueden a su vez contener bloques de código, por ejemplo if, if-else o while. Los diferentes bloques de código se delimitan por el uso de llaves y debemos garantizar el posible y correcto uso de las variables según el ámbito de declaración de las mismas. De esta forma, permitiremos el anidamiento de bloques que deberá respetar estas propiedades.

3. Tipos, Identificadores, Operadores e Instrucciones

Una vez descrita la estructura que tendrán los programas escritos en nuestro lenguaje de programación, pasamos a detallar los elementos que utilizaremos para construir las instrucciones del lenguaje. Empezaremos con los elementos más básicos como las variables y las constantes, llegando hasta la sintaxis de las instrucciones y las funciones.

Tendremos dos conjuntos de constantes que serán los tipos de datos básicos de nuestro lenguaje: Int = $\mathbb{Z}$ y Bool = {True, False}. Estos, junto al tipo Array que se explica a continuación, serán los tipos que tomarán las variables. Dichas variables estarán identificadas por cadenas de caracteres (identificadores) que podrán tener dígitos numéricos, siempre y cuando no estén en primera posición. También usaremos identificadores para nombrar a las funciones.

Podrán crearse listas en forma de Arrays (de una o varias dimensiones). Vamos a imponer la restricción de que todos los elementos de la lista deben de ser del mismo tipo y, además, la longitud de un Array deberá ser indicada en su declaración. Los posibles tipos de un Array son Int, Bool o un Array de un cierto tipo y tamaño. Por tanto, algunos ejemplos de instancias de listas serán [1, 2, 3, 4], [ ], [[1, 2], [3, 4]] y [True, False]. Sin embargo, [1,True], [1,[1,2]] o [[True], [False, False]] no podrán ser construidos con nuestra sintaxis. Adicionalmente, se podrán crear expresiones que contengan a variables, funciones y constantes utilizando operadores infijos o prefijos con prioridades fijadas caso por caso.

Podremos tener comentarios dentro del código, y los identificaremos poniendo // al inicio de los mismos (hasta que haya un salto de línea).

Procedemos a introducir algunos operadores básicos que contendrá nuestro lenguaje. Los operadores asociados a los valor de tipo Int son: + (binario y asociativo por la izquierda), * (binario y asociativo por la izquierda), /(binario y no asociativo), % (binario y no asociativo) y $\wedge$ (binario y no asociativo), que realizarán las operaciones de suma, producto, división entera, módulo y potencia. Además, el operador - estará sobrecargado y podrá usarse como operador binario no asociativo, representando la resta de dos enteros, o como operador unario prefijo, representando el opuesto de un entero. La precedencia será la habitual.

Para los booleanos también contamos con algunos operadores básicos: && (binario y asociativo por la izquierda), || (binario y asociativo por la izquierda) y ! (unario y prefijo), que representan la conjunción, disyunción y negación lógica. La precedencia será la habitual, es decir, la expresión True && False || True se interpreta como (True && False) || True, la expresión !True && False se interpreta como (!True) && False y la expresión !True || False se interpreta como (!True) || False.

Sobre los Arrays tendremos la operación de acceso a un elemento [ ]. Por tanto, dado la instancia a = [1,2,3,4,5] de Array <Int, 5> se tiene que a[0] devuelve 1, a[1] devuelve 2 y así hasta a[4] que devuelve 5. El acceso a índices del array fuera de rango debe generar algún tipo de excepción en tiempo de ejecución.

Todos los tipos de datos anteriormente descritos tienen otros dos operadores binarios de igualdad o desigualdad que devuelven un valor de tipo Bool. Los operadores son: == (binario y asociativo por la izquierda) y != (binario y asociativo por la izquierda). Nótese que la asociatividad solo tiene sentido únicamente para la igualdad o desigualdad booleanas, ya que (1 == 1) == 2 está mal tipado. De igual manera, para los enteros tendremos los operadores de desigualdad: <=, >=, < y >. Todos ellos devuelven un valor booleano.

En esta primera aproximación no tenemos intención de introducir punturos. Sin embargo, podemos presentarlos en caso de que finalmente los incorporaramos. Introducimos el operador *, que devuelve lo apuntado por el puntero. Es un operador unario prefijo que tendrá la mayor precedencia de todas. También debemos introducir variables de tipo puntero a Var donde Var es un tipo de variable construible. De esta forma, podríamos contruir punteros a enteros, punteros a listas de enteros o punteros a listas de punteros a enteros entre otros. Asimismo, la reserva de memoria dinámica se realizaría con la sentencia new, de manera similar a como se hace en C++

Presentamos a continuación las instrucciones de las que dispondrá a priori nuestro lenguaje de programación. No descartamos incluir más instrucciones o ampliar el lenguaje según lo vayamos desarrollando pero, en un principio, nos conformamos con hacer bien lo que sigue. Junto con la descripción semántica en alto nivel de cada instrucción, se aporta un ejemplo de uso para cada una de ellas, que sirve para dar una idea de la sintaxis de las mismas.

- **Instrucción de definición de variables**:

  Permite crear una variable de tipo Int, Bool o Array asignándole un identificador. Terminará cuando se ponga el símbolo de punto y coma, al igual que en el lenguaje C++. Para las listas se deverá indicar su longitud (una constante numérica no negativa) en la declaración y también su tipo, que a su vez será Int, Bool o recursivamente, otro Array. Esto último nos permitirá crear listas de varias dimensiones.

```
Int a;
Bool b;
Array <Int,5> c;
Array <Array <Int,5>,10> d; // d es una tabla de
                            // enteros de 10 filas y 5 columnas
```

- **Instrucción de Asignación**:

  Permite asignar un valor de tipo Int, Bool o Array (devuelto por una expresión, una constante, una función u otra variable) a una variable del mismo tipo que haya sido previamente definida. Por comodidad, permitiremos fusionar instrucciones de definición de variables con instrucciones de asignación para facilitar la creación de variables inicializadas a un cierto valor. Terminará con el símbolo de punto y coma.

```
x = 5;
y = (2 + 3) / 5;
Int z = x + y;
w = [1, 2, z];
```

- **Instrucción de Lectura:**

  Permite leer un valor del tipo Int, Bool o Array que indique el usuario, guardándolo en una variable. El valor leído se extraerá de línea de comandos.

```
read ( x );
```

- **Instrucción de Escritura:**

  Permite escribir un valor del tipo Int, Bool o Array en línea de comandos.

```
print ( True );
print ( x );
print ( [True, True] );
print ( funcion ( x ) );
```

- **Instrucción If:**

  Permite ejecutar un bloque de código supuesto que es cierta una condición booleana, u otra bloque de código distinto supuesto que es falsa. Distinguiremos dos tipos: una instrucción if sin clausula else, y otra instrucción if con clausura else. Es importante destacar que exigiremos el uso de paréntesis para embeber la condición booleana y el uso de llaves para los bloques if y else para evitar ambiguedades.

```
if ( True ) { x = 4; }
if ( x < 5 ) { x = x + 1; } else { x = x - 1; }
```

- **Instrucción While:**

  Permite ejecutar un bloque de código de forma repetida mientras una cierta condición booleana es cierta.

```
while ( 2 + x == 3 ) { x = 2; }
while ( x > 1 ) { x = x + 1; y = x; }
```

- **Instrucción de creación y definición de funciones**:

  Permite crear una función con unos ciertos parámetros (pasados por referencia o por valor), un cierto código y un cierto valor de retorno. En caso de no haber valor de retorno o no haber parámetros de entrada, usaremos void. Además, tendremos una instrucción adicional de return para indicar el valor que se desea retornar. Dicho valor será el resultado de evaluar una expresión. Por defecto, los argumentos se pasarán por valor y para indicar el paso de una variable por referencia se utilizará el símbolo &.

```
def void fun ( Int & p1, Bool p2 ) { if ( p2 ) { print ( p1 ); } }
def Int aux ( void ) { fun (2, True); Int x = 2; return x - 1; }
```

4. Ejemplos Completos de Programas

Presentamos como última sección de este documento una serie de ejemplos de código completos de nuestro lenguaje. Están construidos según las reglas que se han descrito en anteriores secciones y utilizan todo el abanico de instrucciones, operadores y tipos propuestos. Empezamos con un ejemplo sencillo de generación de números de Fibonacci que muestra definiciones de funciones, declaraciones, inicializaciones, instrucciones anidadas if y while, asignaciones, las instrucciones read y print, condiciones...

Listing 1: Generación de números de Fibonacci

```
1  def Int main ( void ) {
2          Int aux = 1;
3          Int fib = 0;
4          Int lim;
5          Int init = 1;
6          read ( lim );
7          if ( lim > 0 ) {
8                  while ( init <= lim ) {
9                       print ( fib );
10                      aux = aux + fib;
11                      fib = aux - fib;
12                      init = init + 1;
13                 }
14         }
15         print(a);
16         return 0;
17 }
```

A continuación, presentamos un ejemplo de ordenamiento de burbuja que va a servirnos para mostrar la creación y acceso a arrays. Además, utilizaremos una función auxiliar que se encarga de intercambiar dos valores pasados por referencia.

Listing 2: Ordenamiento de Burbuja (Bubble Sort)

```
def Int main ( void ) {
        Int i = 1;
        Int n = 100;
        Bool sort = False;
        Array <Int,100> a;
        read ( a );
        while ( i < n && !sort) {
            i = i + 1;
            sort = True;
            Int j = 0;
            while ( j <= n-i ) {
                if ( a[j] > a[j+1] ) {
                    sort = False;
                    intercambiar(a[j],a[j+1]);
                }
            }
        }
        print( a );
        return 0;
}
```

Listing 3: Intercambio de dos valores pasados por referencia a una función

```
def void intercambiar(Int & a, Int & b) {
        Int aux = a;
        a = b;
        b = aux;
}
```

El siguiente ejemplo es un contador de años bisiestos que muestra diversas operaciones booleanas y aritméticas.

Listing 4: Contador de años bisiestos

```
def Int main ( void ) {
        Int count = 0;
        Int ini = 104;
        Int end = 643;
        while ( ini <= end ) {
            if ( bisiesto(ini) ) {
                count = count + 1;
            }
            ini = ini + 1;
        }
        print(count);
        return 0;
}
```

Listing 5: Identificador de años bisiestos

```
def Bool bisiesto (Int & a) {
        Bool dev;
        if ((a%4==0) && ((a%100!=0) || (a%400==0))) {
            dev = True;
        } else {
            dev = False;
        }
        return dev;
}
```

Por último, terminamos con un algoritmo escrito en nuestro lenguaje que resuelve el problema 121, titulado Chicles de regalo, de la famosa página de problemas de programación Acepta el Reto.

Listing 6: Solución al Problema 121 (Acepta el Reto) (Chicles de regalo)

```
def int main ( void ) {
    Int numRegalo;
    Int numE;
    Int numChicles;
    Int numComidos;
    Int numER;
    while ( True ) {
        numComidos = 0;
        numEnvoltoriosR = 0;
        read ( numEnvoltorios );
        if (numEnvoltorios == 0) {
            return 0;
        }
        read ( numRegalo );
        read ( numChicles );
        if ((numRegalo >= numE) && (numChicles >= numE)) {
            print ( 0 );
        } else {
            while(numChicles > 0) {
                numComidos = numComidos + numChicles;
                numER = numER + numChicles;
                numChicles = 0;
                numChicles = numER / numE * numRegalo;
                numER = numER - numE / numE * numE;
            }
            print ( numComidos );
            print ( numER );
        }
    }
    return 0;
}
```