Diunisio: Manual de usuario

Andrés Felipe De Orcajo Vélez
Jorge Eduardo Ortiz Triviño

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, Colombia
2017

Índice

1. Introducción

1.1. Programación Imperativa

La programación imperativa es un paradigma de programación de computadores, que describe la programación en términos del estado del programa y sentencias que cambian dicho estado. Los programas imperativos son un conjunto de instrucciones que le indican al computador cómo realizar una tarea. Por ejemplo, cada paso en una receta es una instrucción.[1]
Se basa en tres conceptos:

- **Secuencia:** donde se ejecutan sentencias ordenadamente unas después de otras.
- **Selección:** donde se bifurca el código, permite elegir el camino a tomar por el algoritmo en ejecución.
- **Iteración:** donde se repiten las sentencias agrupadas por un iterador, hasta que se cumpla una condición de finalización.

1.2. Descripción general

Diunisio es un lenguaje de programación imperativo en español desarrollado con la librería ANTLR; que permite construir algoritmos con sintaxis flexible, fácil de entender, de bajo tipado y con funcionalidades añadidas.

2. Descripción del modelo de programación

El algoritmo es programado usando caracteres en formato de texto plano usando los caracteres del conjunto ASCII. Los caracteres válidos en el algoritmo son los siguientes:

- caracteres alfabéticos:

  ```
  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 _
  ```

- caracteres numéricos:

  ```
  0 1 2 3 4 5 6 7 8 9
  ```

- caracteres especiales:

  ```
  ! & ( ) * , - . : ; < = > / % + [ ] ^ { | }
  ```

- caracteres de espacios en blanco:

  ```
  espacio nuevaLinea tabulación
  ```

Dentro de los literales y comentarios cualquier caracter ASCII (excepto caracteres de control) son válidos.
Los caracteres de espacios en blanco no son significantes. Pueden ser usados libremente entre unidades léxicas para mejorar la legibilidad del algoritmo. También son usadas para separar unidades léxicas unas de otras si no hay otra forma de hacerlo.
Sintácticamente el algoritmo es una secuencia de unidades léxicas en las siguientes categorías:

- nombres de variables;
- números;
- cadenas de caracteres;
- palabras reservadas;
- delimitadores y operadores;
- comentarios.

Las unidades léxicas del lenguaje son presentadas a continuación.

2.1. Nombres de variables

Un nombre de variable consiste en caracteres alfanuméricos, el primero debe ser alfabético. Todos los nombres de variables son distintos (distingue mayúsculas y minúsculas).

Ejemplos:

```
variable123
Estatura_promedio
_VALOR
```

Los nombres de variables son usados para identificar variables (números, parámetros, conjuntos).
Todos los nombres de variables deben ser únicos dentro del alcance, donde sean válidos.

2.2. Números

Un número tiene la forma *nnEsxx*, donde *nn* es un número con punto decimal opcional, *s* es el signo + o -, *xx* es un exponente decimal. La letra *E* también puede ser *e*.

Ejemplos:

```
123
3.14159
56.E+5
.78
123.456e-7
```

Los números son usados para representar cantidades numéricas.

2.3. Números complejos

Un número complejo tiene la forma *nnEsxxEsnnEsxxi*, donde *nn* es un número con punto decimal opcional, *s* puede ser el signo - o + (excepto cuando es el signo del exponente), *xx* es un exponente decimal. La letra *E* también puede ser *e*.

Ejemplos:

```
123+i
3.14159-2i
56.E5+2.2i
.78-.9E5i
123.456e-7+i
```

Los números complejos son usados para representar cantidades numéricas en el plano complejo.

2.4. Cadena de caracteres

Una cadena de caracteres es una secuencia de caracteres arbitrarios encerrados por comillas dobles.

Ejemplos:

```
"Esta es una cadena de caracteres"
"Hola "
```

Las cadenas de caracteres son usadas para representar cantidades simbólicas.

2.5. Palabras Reservadas

Una palabra reservada es una secuencia de caracteres alfabéticos y posiblemente un caracter especial.
Todas las palabras reservadas corresponden a una categoría: palabras reservadas, las cuales pueden ser usadas como nombres de variables, y palabras no reservadas, las cuales son reconocidas por el contexto y por eso pueden ser usadas como nombres de variables.
Las palabras reservadas son las siguientes:

```
ALGORTIMO    booleano  cadena  caso       decimal  def    defecto   entero
entonces     falso     hacer   mientras   nulo     para   retornar  romper
seleccionar  si        si_no   verdadero
```

Las palabras no reservadas son presentadas en las siguientes secciones.
Todas las palabras reservadas tienen un significado específico, el cual será explicado en las construcciones sintácticas correspondientes, donde las palabras reservadas son usadas.

2.6. Delimitadores y operadores

Un delimitador es o un caracter especial simple o una secuencia de dos caracteres especiales como se muestran a continuación:

```
! != && ( ) * , - . : ; < <= = == > >= / % + [ ] ^ { || }
```

Si el delimitador consiste de dos caracteres, no debe haber espacio entre los caracteres.
Todos los delimitadores tienen un significado específico, el cual será explicado en las correspondientes construcciones sintácticas, donde los delimitadores son usados.

2.7. Comentarios

Para propósitos de documentación del algoritmo puede contener comentarios, que tienen dos formas. La primera forma es la de comentarios de línea simple, que empieza con el caracter `#` y se extiende hasta el fin de línea. La segunda forma es la de comentario multilínea, que es una secuencia de caracteres encerrados dentro de `/*` y `*/`.

Ejemplos:

```
i = 0; #Esto es un comentario
/* Esto es otro comentario */
```

Los comentarios son ignorados por el compilador y pueden aparecer en cualquier lado del algoritmo, donde los espacios en blanco están permitidos.

3. Expresiones

Una expresión es una regla para calcular un valor. En el algoritmo las expresiones son usadas como constituyentes de ciertas sentencias.
En las expresiones generales consiste en operandos y operadores.
Dependiendo del tipo del valor resultante todas las expresiones caen dentro de una de las siguientes categorías:

- expresiones numéricas;
- expresiones simbólicas;
- expresiones lógicas.

3.1. Expresiones numéricas

Una expresión numéricas es una regla de computación para un valor numérico simple representado como un número de punto flotante.
La expresión numérica primaria puede ser un número, un parámetro, una referencia a una función predefinida, u otra expresión numérica encerrada en paréntesis.
Ejemplos:

```
1.23
j
tiempo
mat[1][2]
abs(mat[1][2])
(mat[1][2] * .5 * j)
```

Más expresiones numéricas conteniendo dos o más expresiones numéricas primarias pueden ser construidas usando ciertos operadores aritméticos.
Ejemplos:

```
j+1
2 * mat[j][i-1] - b[1][2]
```

3.1.1. Referencia de funciones

En Diunisio existen las siguientes funciones predefinidas las cuales pueden ser usadas en expresiones numéricas:

Función	Descripción
`sen(x)`	Seno de `x`
`cos(x)`	Coseno de `x`
`tan(x)`	Tangente de `x`
`sec(x)`	Secante de `x`
`csc(x)`	Cosecante de `x`
`ctg(x)`	Cotagente de `x`
`subcadena(cadena, inicio, fin)`	Subcadena de `cadena`
`leerTeclado(tipo, mensaje)`	Lectura de teclado
`leerArchivo(directorio, fila)`	Retorna el dato del archivo en la fila elegida
`tamanoArchivo(directorio)`	Retorna el número de filas del archivo
`tamano(conjunto)`	Retorna el tamaño de un conjunto
`aleatorio(inferior, superior)`	Genera un número aleatorio uniforme dentro del intervalo dado

Los argumentos de todas las funciones predefinidas deben ser expresiones numéricas a menos que sea un directorio.
El valor resultante de una expresión numérica, la cual es una referencia de función, es el resultado de aplicar la función a su(s) argumento(s).
Nota: las funciones trigonométricas usan números radianes.

3.1.2. Expresiones con paréntesis

Cualquier expresión numérica puede ser encerrada en paréntesis que sintácticamente lo hacen una expresión numérica primaria.
Los paréntesis pueden ser usados en expresiones numéricas, como en el álgebra, para especificar el orden deseado del cual las operaciones se van a desarrollar. Donde los paréntesis son usados, la expresión dentro de los paréntesis es evaluada antes de que el valor resultante sea usado.
El valor resultante de las expresiones con paréntesis es la misma que el valor de la expresión encerrada dentro de los paréntesis.

3.1.3. Operadores aritméticos

En Diunisio existen los siguientes operadores aritméticos, los cuales pueden ser usados en expresiones numéricas:

Operación	Descripción
`-n`	Operador unario negativo
`+n`	Operador unario positivo
`n + m`	Suma
`n - m`	Resta
`n * m`	Producto
`n / m`	División
`n % m`	Módulo
`n ^ m`	Potenciación

donde `n` y `m` son expresiones numéricas.
Si la expresión incluye más que un operador aritmético, todos los operadores son aplicados de izquierda a derecha de acuerdo a la jerarquía de operadores (mirar a continuación) con la única excepción de que los operadores de exponenciación son aplicados de derecha a izquierda.
El valor resultante de la expresión, la cual contiene operadores aritméticos, es el resultado de aplicar los operadores a sus operandos.

3.1.4. Jerarquía de operaciones

La siguiente lista muestra la jerarquía de los operadores en expresiones numéricas:

Operación	Jerarquía
Evaluación de funciones	1º
Exponenciación	2º
Suma y resta unarias	3º
Multiplicación y división	4º
Suma y resta	5º

La jerarquía es usada para determinar cuál de dos operaciones consecutivas es aplicada primero. Si el primero operador es mayor o igual al segundo, el primer operador es aplicado. Si no, el segundo operador es comparado con el tercero, etc. Cuando el fin de la expresión es alcanzado, todas las operaciones restantes son ejecutadas en el orden inverso.

3.2. Expresiones simbólicas

Una expresión simbólica es una regla de computación de un solo valor simbólico representado como una cadena de caracteres.
La expresión simbólica primaria puede ser una cadena de caracteres, un parámetro, una referencia de función predefinida, u otra expresión simbólica encerrada entre paréntesis.
También es permitido usar una expresión numérica como función simbólica primaria, caso en el cual el valor resultante de una expresión numérica es automáticamente convertido a un tipo simbólico.

Ejemplos:

```
"Mayo 2017"
j
subcadena(1,5)
```

Más expresiones simbólicas conteniendo dos o más expresiones simbólicas primarias puede ser construida usando el operador de concatenación.

Ejemplos:

```
"abc[" + i + "," + j + "]"
"desde " + ciudad[i] + " a " + ciudad[j]
```

Los principios de evaluación de expresiones simbólicas son complementariamente análogas a las dadas para expresiones numéricas (mirar arriba).

3.2.1. Referencia de funciones

En Diunisio existen las siguientes funciones predefinidas las cuales pueden ser usadas en expresiones simbólicas:

Operación	Jerarquía
subcadena(cadena, inicio, fin)	Subcadena de `r`

El primer argumento de subcadena debería ser una expresión simbólica mientras que el segundo argumento debería ser una expresión numérica.
El valor resultante de la expresión simbólica, la cual es una referencia de una función, es el resultado de aplicar la función a sus argumentos.

3.2.2. Operadores simbólicos

Actualmente en Diunisio existe un sólo operador simbólico:

Operación	Jerarquía
`s + t`	Concatenación

donde `s` y `t` son expresiones simbólicas. Éste operador significa la concatenación de sus dos operandos simbólicos, los cuales son cadenas de caracteres.

3.2.3. Jerarquía de operadores

La siguiente lista muestra la jerarquía de operadores en las expresiones simbólicas:

Operador	Jerarquía
Evaluación de operaciones numéricas	1º-6º
Concatenación	7º

Esta jerarquía tiene el mismo significado como se explicó anteriormente para expresiones numéricas.

3.3. Expresiones de conjuntos

Una expresión de conjuntos es una regla de cálculo de un conjunto elemental, por ejemplo una colección de n-tuplas, donde los componentes de las n-tuplas son cantidades numéricas y simbólicas.
La expresión de conjuntos básica puede ser una definición de conjunto, u otra expresión de conjuntos encerrada entre paréntesis.

Ejemplos:

```
{1,2}
{{1,2},{3,4}}
{{{1},{2,3}}}
```

3.3.1. Expresiones con paréntesis

Cualquier expresión de conjuntos puede estar encerrada entre paréntesis que sintácticamente lo hacen una expresión de conjunto básica.
Los paréntesis pueden ser usados en expresiones de conjuntos, como en el álgebra, para especificar el orden deseado en el cual las operaciones van a ser aplicadas. Donde los paréntesis son usados, la expresión dentro de los paréntesis es evaluada antes de que el valor resultante es usado.
El valor resultante de la expresión con paréntesis es la misma que la evaluada de la expresión encerrada por los paréntesis.

3.3.2. Operaciones de conjuntos

En Diunisio existen las siguientes operaciones, las cuales pueden ser usadas en expresiones de conjuntos:

Operación	Descripción
`x + y`	Suma
`x - y`	Resta
`x * y`	Producto
`x / y`	División

donde `x` e `y` son expresiones de conjuntos, los cuales deberían definir conjuntos de dimensión idéntica.
Si la expresión incluye más de un conjunto de operadores, todos los operadores son aplicados de izquierda a derecha de acuerdo a la jerarquía de operadores (mirar a continuación).

El valor resultante de la expresión, el cual contiene un conjunto de operadores, es el resultado de aplicar los operadores a sus operandos.
La dimensión del conjunto resultante, por ejemplo, la dimensión de n-tuplas, de las cuales el conjunto resultante se compone, es el mismo de la dimensión de los operandos.
Nota: Si las dimensiones de las matrices no coinciden, el operador de suma une los conjuntos. El operador de producto realiza producto cruz cuando son matrices y producto punto cuando son vectores.

3.3.3. Jerarquía de los operadores

La siguiente lista muestra la jerarquía de los operadores en expresiones de conjuntos:

Operación	Jerarquía
Evaluación de operaciones numéricas	1º-6º
Evaluación de operaciones simbólicas	7º-9º
Producto y división de conjuntos	10º
Suma y resta de conjuntos	11º

La jerarquía tiene el mismo significado de lo explicado anteriormente para las expresiones numéricas.

3.4. Expresiones lógicas

Una expresión lógica es una regla de cálculo de un único valor lógico, el cual puede o ser verdadero o falso.
La expresión lógica más simple puede ser una expresión numérica, una expresión relacional, u otra expresión lógica encerrada entre paréntesis.

Ejemplos:

```
i+1
a[1] < 8
a[2] != 5
falso && !verdadero
```

Expresiones lógicas más generales conteniendo dos o más expresiones lógicas simples pueden ser construidas usando ciertos operadores lógicos.

Ejemplos:

```
!(a[i] < 8 || b[j] >= 9) && i < 9
```

3.4.1. Expresiones numéricas

El valor resultante de la expresión lógica simple, la cual es una expresión numérica, es verdadero, si el valor resultante de la expresión no es cero. En otro caso el valor resultante de la expresión numérica es falsa.

3.4.2. Operadores relacionales

En Diunisio existen los siguientes operadores lógicos, los cuales pueden ser usados en expresiones lógicas:

Operación	Descripción
`x < y`	Menor que
`x <= y`	Menor o igual que
`x == y`	Igual
`x >= y`	Mayor o igual que
`x > y`	Mayor que
`x != y`	Distinto de

donde `x` e `y` son expresiones numéricas o simbólicas.
Como los operadores relacionales listados anteriormente tienen su significado matemático convencional. El valor resultante es verdadero, si la relación correspondiente es satisfecha por sus operandos, en otro caso falso. (Notar que los valores simbólicos son ordenados lexicográficamente, y cualquier valor numérico precede cualquier valor simbólico.

3.4.3. Expresiones con paréntesis

Cualquier expresión lógica puede ser encerrada en paréntesis que sintácticamente lo hacen una expresión lógica simple.
Los paréntesis pueden ser usados en expresiones lógicas, como en el álgebra para especificar el orden deseado en el cual los operadores se aplicarán. Donde los paréntesis son usados, la expresión dentro de los paréntesis es evaluada antes de que el valor resultante es usado.
El valor resultante de la expresión con paréntesis es la misma que el valor de la expresión encerrada dentro de los paréntesis.

3.4.4. Operadores lógicos

En Diunisio existen los siguientes operadores lógicos, los cuales pueden ser usados en expresiones lógicas:

Operación	Descripción
`!x`	Negación
`x && y`	Conjunción
`x || y`	Disjunción

donde `x` e `y` son expresiones lógicas.
Si la expresión incluye más de un operador lógico, todos los operadores son aplicados de izquierda a derecha de acuerdo a la jerarquía de los operadores (mirar a continuación). El valor resultante de la operación, el cual contiene operadores lógicos, es el resultado de aplicar los operadores a sus operandos.

3.4.5. Jerarquía de operaciones

La siguiente lista muestra la jerarquía de los operadores en las expresiones lógicas:

Operación	Jerarquía
Evaluación de operaciones numéricas	1º-6º
Evaluación de operaciones simbólicas	7º
Evaluación de operaciones de conjuntos	8º
Operaciones relacionales	9º
Negación	10º
Conjunción	11º
Disyunción	12º

La jerarquía tiene el mismo significado que fue explicado anteriormente para expresiones numéricas.

4. Sentencias

Las sentencias son unidades básicas en la descripción del algoritmo. En Diunisio todas las sentencias están divididas en dos categorías: sentencias de declaración y sentencias funcionales.
Las sentencias de declaración (sentencia de conjunto, sentencia de parámetros, sentencia de variable) son usados para declarar los objetos del algoritmo de ciertos tipos y definir ciertas propiedades de esos objetos.

4.1. Sentencia ALGORITMO

```
ALGORITMO nombre ( param, ..., param ) : {
        sentencias
}
.
```

`nombre` es un nombre simbólico del algoritmo;
`param, ..., param` son parámetros opcionales de entrada.

Ejemplos:

```
ALGORITMO calculos():{
        imprimirPantalla(2+2);
}
.
```

4.2. Declaración de variables

```
nombre = valor;
```

`nombre` es un nombre de variable;
`valor` puede ser una expresión numérica, lógica, de conjunto, simbólica.

Ejemplos:

```
a = 1;
b = i;
c = "Cantidad";
e = verdadero;
```

4.3. Sentencia si

```
si ( condición ) entonces {
        sentencias
}
si_no si ( condición ) {
        sentencias
}
...
si_no si ( condición ) {
        sentencias
}
si_no {
        sentencias
}
```

`condición` es la condición que se evaluará para determinar si se elige la opción o se continúan recorriendo las demás condiciones;
`si_no si ... si_no si` es la serie de condiciones opcionales alternativas al primer `si`;
`si_no` es la condición por defecto si ninguna de las demás se llegara a cumplir.
Nota: los paréntesis son opcionales junto con la palabra entonces.

Ejemplos:

```
si a < 0 { imprimirPantalla("negativo"); }
si b { a = a * 5; } si_no si (a == 1) entonces { b = b / 5; }
si falso entonces { } si_no si verdadero { }
```

4.4. Sentencia seleccionar

```
seleccionar variable {
    caso valor: sentencias romper;
    ...
    caso valor: sentencias romper;
    defecto: sentencias
}
```

`variable` es un valor simbólico o numérico que se tomará como referencia para elegir el `caso` que coincida en su valor con ésta variable;
`valor` es el valor que se comparará con la `variable` para determinar si el caso es elegido o no;
`caso ... caso` permite añadir cualquier cantidad de casos, que pueden no definirse pero el caso por `defecto` siempre estará presente.

Ejemplos:

```
seleccionar a {
    caso 1: imprimirPantalla("solo hay uno"); romper;
    caso 2: imprimirPantalla("quedan dos"); romper;
    defecto: imprimirPantalla("no alcanza");
}
```

4.5. Sentencia para

```
para ( var, ..., var; condición; oper, ..., oper) {
        sentencias
}
```

`var, ..., var` es un listado de declaración de variables;
`condición` es la condición que se evaluará por cada iteración del ciclo;
`oper, ..., oper` es un listado de operaciones que modifiquen variables para proceder a terminar el ciclo.
Nota: los paréntesis son opcionales.

Ejemplos:

```
para a = -1; a < 5; a=a+1 {
        imprimirPantalla(a);
}
para (i = 0, j = 0; i < 5; i=i+1, j = i-1) {
        imprimirPantalla(mat[i][j]);
}
```

4.6. Sentencia mientras

```
mientras ( condicion ) {
        sentencias
}
```

`condición` es la condición que se evaluará por cada iteración del ciclo.
Nota: los paréntesis son opcionales.

Ejemplos:

```
mientras a>2 {
        a = a-1;
        imprimirPantalla(sen(a));
}
mientras a + b < c {
        imprimirPantalla("sumando a y b");
        c = c - 1;
}
```

4.7. Sentencia hacer_mientras

```
hacer {
        sentencias
} mientras ( condición )
```

`condición` es la condición que se evaluará al finalizar cada iteración para determinar la continuidad del ciclo.
Nota: los paréntesis son opcionales.

Ejemplos:

```
hacer {
        a = a+1;
        imprimirPantalla(a);
} mientras a < 5
```

4.8. Procedimiento

```
def nombre ( param, ..., param ) {
        sentencias
}
```

`nombre` determina el nombre con el que se almacenará el procedimiento;
`param, ..., param` es una serie de parámetros que recibirá el procedimiento al momento de ejecutarse.

Ejemplos:

```
def concat (nombre) {
        imprimirPantalla("Hola " + nombre);
}
a = concat("Andrés");
```

4.9. Función

```
def tipo nombre ( param, ..., param ) {
        sentencias
}
```

`nombre` determina el nombre con el que se almacenará la función;
`param, ..., param` es una serie de parámetros que recibirá la función al momento de ejecutarse.

Ejemplos:

```
#Definición recursiva del factorial de 'y'
def entero g(y){
        res = 1;
        si y < 2 {
                res = 1;
        }
        si_no {
                res = y * g(y-1);
        }
        retornar res;
}
e = g(15);
imprimirPantalla ("Factorial recursivo de 15 = " + e);
```

5. Requerimientos

A continuación se explica el procedimiento de instalación junto con los requisitos del sistema para poder ejecutar el algoritmo correctamente.

5.1. Software

Para la correcta ejecución de los algoritmos sobre el lenguaje Diunisio, es necesaria la instalación del siguiente software, junto con un sistema operativo, ya sea Windows o alguna distribución de Linux:

- Java Development Kit (JDK) 7.
- Librería AntLR v4.

5.2. Hardware

Los requerimientos de hardware están dados por el sistema operativo y por el entorno de desarrollo de Java.

6. Instalación

6.1. Windows

6.1.1. Instalación del JDK

Descargar el JDK de la página de `http://www.oracle.com/technetwork/es/java/javase/downloads/index.html`, luego se instala. Se debe editar las variables del entorno de Windows, añadiendo a la variable PATH el directorio de instalación del JDK:

```
PATH= ... ;C:\Program Files\Java\jdk1.8.0_112\bin
```

6.1.2. Descarga de ANTLR

Descargar la librería de ANTLR desde `http://www.antlr.org/`

6.1.3. Configuración adicional

Se debe comprobar la correcta instalación del entorno de desarrollo de Java ejecutando la siguiente instrucción en la terminal:

```
javac -version
```

Creamos un directorio en la Unidad de Disco Local C: llamado *ANTLR*, y dentro:

1. Creamos un archivo *antlr4.bat* y en su contenido ponemos:

```
java org.antlr.v4.Tool %*
```

2. Creamos un archivo *grun.bat* y en su contenido ponemos:

```
java org.antlr.v4.gui.TestRig %*
```

3. Copiamos ahí la librería de ANTLR previamente descargada.

Añadimos la siguiente variable del entorno:

```
CLASSPATH = .;C:\ANTLR\antlr-4.7-complete.jar
```

6.2. Linux

6.2.1. Instalación del JDK

Para obtener el kit de desarrollo de Java, debemos ejecutar desde el terminal el siguiente comando como superusuario:

```
sudo apt-get install openjdk-7-jdk
```

6.2.2. Descarga de ANTLR

La librería de ANTLR se puede descargar de la página: `http://www.antlr.org/`

6.2.3. Configuración adicional

Para que el JDK al hacer la ejecución del intérprete, pueda completar correctamente el proceso de compilación, deben definirse las variables del entorno: *CLASSPATH*; pero también para facilitar la inserción de parámetros de la ejecución de ANTLR, se deben crear también otras dos variables: *antrl4* y *grun*. Las cuales se deben ejecutar directamente en la terminal. La estructura de la variable *CLASSPATH* es:

```
export CLASSPATH=".:/directorio/antlr-4.7-complete.jar:$CLASSPATH"
```

Un ejemplo de *CLASSPATH*:

```
export CLASSPATH=".:/home/usuario/Escritorio/antlr-4.7-complete.jar:$CLASSPATH"
```

Luego se añaden estos alias, también desde el terminal:

```
alias antlr4='java org.antlr.v4.Tool'
alias grun='java org.antlr.v4.gui.TestRig'
```

Rutas del entorno para el compilador de Java

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk
export PATH=$PATH:/usr/lib/jvm/java-7-openjdk/bin
```

Se debe comprobar la correcta instalación del entorno de desarrollo de Java ejecutando la siguiente instrucción en la terminal:

```
javac -version
```

7. Ejecución

Para poder hacer uso de la herramienta, es necesario el uso de una consola (o un terminal) del sistema operativo para ejecutar la clase *Main.java* contenida en el directorio de Diunisio. Primero se deben compilar todas las clases del directorio, haciendo uso del JDK instalado previamente:

```
javac *.java
```

A continuación, situados en el mismo directorio, ejecutamos la clase compilada Main así:

```
java Main
```

La consola (o terminal) quedará a la espera para recibir la introducción del algoritmo elaborado con la sintaxis de Diunisio, por lo tanto debemos, o introducir el programa manualmente, o pegarlo directamente:
Cuando se haya introducido completamente el algoritmo, debemos presionar *ENTER* y luego *CTRL+D* (o *CTRL+Z* desde Windows) para informarle a la consola (o terminal) de que hemos terminado la introducción del contenido del programa, finalmente se observarán los resultados de la ejecución del algoritmo introducido previamente.

Nota: Es posible ejecutar un código previamente almacenado en un archivo, ejecutando el siguiente comando:

```
java Main archivo.txt [parámetros del algoritmo]
```

8. Ejemplos

8.1. Fibonacci

8.2. Cálculo de Fibonacci iterativo:

```
ALGORITMO fibonacciIterativo(NULL):
{
        def entero g(n){
                x = 0;
                y = 1;
                z = 1;
                para i=0; i<n; i=i+1 {
                        x = y;
                        y = z;
                        z = x + y;
                }
                retornar x;
        }
        e = g(8);
        imprimir (e);
}
.
```

8.3. Cálculo de Fibonacci recursivo:

```
ALGORITMO fibonacciRecursivo(NULL):
{
        def entero g(y){
                c = b;
                b = a + b;
                a = c;
                res = 0;
                si y < 1 {
                        b = 0;
                }
                si_no si y > 2{
                        res = g(y-1);
                }
                retornar b;
        }
        a = 0;
        b = 1;
        c = a;
        e = g(10);
        imprimir (e);
}
.
```

Referencias

[1] Programación imperativa. (n.d.). En Wikipedia. Recuperado el 15 de enero de 2017, de https://es.wikipedia.org/wiki/Programaci%C3%B3n_imperativa