Universidad de Córdoba

Escuela Politécnica Superior de Córdoba

Intérprete de pseudocódigo en español: interpreter

Procesadores de Lenguajes. Computación.

Grado Ingeniería Informática. Tercer Curso. Segundo Cuatrimestre. Curso 2022-2023

Autor:
Carlos Lucena Robles

Córdoba, Junio 2023

Índice

Índice de Tablas

Índice de Figuras

Lista de Código

1. Introducción

Toda creación de programas informáticos en lenguajes de alto nivel parte codificando el programa en un fichero usando una sintaxis, semántica y palabras que varían entre lenguajes de programación. Para que el ordenador pueda ejecutar las instrucciones del código fuente debe haber una traducción a un lenguaje que entienda el ordenador.

Los programas encargados de la traducción son los compiladores e intérpretes: el primero traduce el código fuente completo y lo convierte en código máquina (crea un programa ejecutable) mientras que el segundo realiza la traducción instrucción por instrucción y no guarda el resultado de la traducción.

Nuestro objetivo es crear un intérprete de pseudocódigo en español por lo que necesitamos definir el idioma, las reglas sintácticas y semánticas que regirán nuestro lenguaje.

La creación de un lenguaje se basa en el uso conjunto de analizadores léxicos y sintácticos. Usaremos como analizador léxico la herramienta Flex y para el análisis sintáctico GNU Bison.

De forma resumida, Flex escaneará completamente el código fuente generando tokens que usará Bison para comprobar que se satisfacen las reglas sintácticas del lenguaje y generará el AST (Abstract Syntax Tree).

Para empezar a, nuestro lenguaje de pseudocódigo será de alto nivel, estructurado y débilmente tipado;

2. Lenguaje de pseudocódigo

2.1. Componentes léxicos. Tokens

2.1.1. Palabras reservadas. *Keywords*

Las palabras reservadas tienen un significado especial, es decir, son usadas por sentencias o expresiones y son parte de la sintaxis del lenguaje.

Nótese que se ha introducido a los operadores lógicos y el nombre de las funciones built-in como *keywords*. (Véase Section 2.1.5 para los operadores lógicos).

Las palabras clave referidas como comandos especiales hace referencia a «sentencias/funciones» del lenguaje encargadas del manejo de pantalla. Su uso puede estar más enfocado al manejo interactivo del programa pero deben marcarse como palabras clave por su significado especial.

Características:

- No pueden usarse como identificadores.
- No se distinguirá entre mayúsculas y minúsculas.

Novedades: `const inf -inf tan to_degrees to_radians style`

Tabla 2.1: *Keywords.* Sentencias

`print`	`print_string`	`read`	`read_string`
`if`	`then`	`else`	`end_if`
`case`	`value`	`default`	`end_case`
`while`	`do`	`end_while`	`repeat`
`until`	`for`	`from`	`to`
`step`	`end_for`	`const`	

Tabla 2.2: *Keywords.* Constantes lógicas y numéricas

`pi`	`e`	`gamma`	`deg`	`phi`
`inf`	`-inf`	`true`	`false`	

Tabla 2.3: *Keywords.* Operadores lógicos

`or`	`and`	`not`

Tabla 2.4: *Keywords.* Funciones built-in

`sin`	`cos`	`tan`	`atan`
`exp`	`sqrt`	`integer`	`abs`
`log`	`log10`	`to_degrees`	`to_radians`
`random`	`atan2`		

Tabla 2.5: *Keywords.* Comandos especiales

`clear_screen`	`place`	`style`

2.1.2. Identificadores

Serán los nombres de las variables y constantes que declare el programador. Servirán como referencia a ellos para operar en expresiones y/o sentencias.

Características:

- Compuestos por letras, dígitos y el subrayado.
- Deben comenzar por una letra.
- No pueden acabar con el símbolo de subrayado ni tener dos subrayados seguidos.
- No se distinguirá entre mayúsculas y minúsculas.

Ejemplos válidos:

- dato o dAtO o DATO
- dato_1
- dato_1_a

Ejemplos NO válidos:

- _dato
- dato_
- dato__1

2.1.3. Números

Los números admitidos son enteros, reales de punto fijo y reales con notación científica. (Véase Section 2.1.5 para los números negativos o positivos).

Características:

- Todos los números son tratados igual (no se distinguen).
- Como delimitador decimal se usa el punto.
- La exponenciación es representada por `E` o `e`

Ejemplos válidos:

- 50
- 50.34
- 0.50 o .50
- 50.0 o 50.
- 4E2 o 4e2
- 2e3 o 2e+3 o 2.e3 o 2.e+3
- 2e-3 o 2.e-3
- .2e2 o .2e+2

Ejemplos NO válidos:

- e2
- .e2
- 2e.2
- 2e2.0
- 2e2e2

2.1.4. Cadenas

Las cadenas son una serie de caracteres alfanuméricos delimitados por comillas simples.

Características:

- Las comillas exteriores no forman parte de la cadena.
- Las cadenas contienen como delimitador final el símbolo '\0'
- No se permiten cadenas anidadas. Para incluir la comilla en la cadena se usa \'.
- Se interpreta el salto de linea (\n) y el tabulador (\t).

Ejemplos válidos:

- 'Esto es una cadena'
- 'Salto de linea \n'
- 'Tabulador \t'
- 'Ejemplo de cadena con \' comillas \' simples'

Ejemplos NO válidos:

- Cadena sin comillas
- 'Cadena sin terminar
- Cadena sin abrir'
- 'Cadena 'dentro de' otra'

2.1.5. Operadores

Los operadores son símbolos que junto a variables, constantes, números o cadenas forman expresiones que producen resultados.

A continuación, se presentan disintos tipos de operadores aritméticos, relaciones, lógicos, alfanuméricos o de asignación. Nótese que en la Section 2.1.1 se han presentado los operadores lógicos como *keywords* porque asi no pueden ser asignados como identificadores.

Características:

- Solo hay dos operadores numéricos aritméticos unarios: `+` y `-`, son usados para escribir un número de forma positiva o negativa. Nótese que son el mismo símbolo para la operación de suma y resta respectivamente.
- Los operadores aritméticos solo operan con números, el alfanumérico con cadenas y los operadores lógicos con constantes lógicas.
- Los operadores relaciones sirven tanto para números, cadenas o constantes lógicas, pero los dos operandos deben ser del mismo tipo. El resultado al evaluar la expresión es una constante lógica.
- Los operadores de asignación cambian el valor de la variable (identificador) al valor deseado.
- El operador de incremento y decremento pueden usarse tanto de forma prefija somo sufija.
- Todos los operadores se pueden usar de forma conjunta y mezclada según las reglas de cada operador.

Tabla 2.6: *Operadores aritméticos*

Suma/Positivo	+
Resta/Negativo	-
Multiplicación	*
División	/
División entera	//
Potencia	^
Módulo	*
Incremento	++
Decremento	--

Tabla 2.7: *Operadores relaciones*

Menor que	<
Menor o igual que	<=
Mayor que	>
Mayor o igual que	>=
Igual que	=
Distinto que	<>

Tabla 2.8: *Operadores de asignación*

Asignación	:=
Asignación de suma	+:=
Asignación de resta	-:=
Asignación de multiplicación	*:=
Asignación de división	/:=
Asignación de división entera	//:=
Asignación de potencia	^:=
Asignación de módulo	%:=

Tabla 2.9: *Operador de alfanumérico*

Concatenación	||

2.1.6. Constantes

Las constantes son variables que una vez asignadas con un valor (véase operador de asiganción en la Section 2.1.5) no pueden cambiar su valor o tipo durante la ejecución del programa.

El lenguaje ya contiene constantes numéricas y lógicas por defecto (véase Section 2.1.1) pero el programador puede definirlas en los programas.

Características:

- Las constantes pueden ser de tipo numérica, lógica o cadena.
- Pueden operarse con ellas como cualquier otra variable según las restricciones del operador y/o sentencia.
- No pueden cambiar su nombre o valor de ninguna forma, por tanto, son las únicas variables que no cambian de tipo.

2.1.7. Comentarios

Conjunto de caracteres alfanuméricos delimitados por símbolos especiales que no serán anlizados ni ejecutados. Los comentarios pueden ser de una linea o varias.

Características:

- No se permite comentarios de varias lineas anidados.
- Se permite la repetición de los símbolos < o >.

Tabla 2.10: ***Comentario de una linea***

Comentario de una linea	#
Comentario de varias lineas	<< >>

2.1.8. Signos de puntuación

Los símbolos siguientes tienen un significado especial dentro del contexto adecuado y no pueden ser usados de otro modo.

Tabla 2.11: ***Signos de puntuación***

Punto y coma (Fin de sentencia)	;
Dos puntos (Sentencia case)	:
Coma (Separador argumentos funciones)	,
Paréntesis (Condiciones, etc)	( )
Llaves (Sentencia bloque)	{ }

2.2. Sentencias

2.2.1. Asignación

Como su nombre indica, permite asignar a una variable (identificador) un valor numérico, lógico o cadena ya sea directamente, con el resultado de una operación, el valor devuelto por una función o por otra variable.

Características:

- La sentencia de asignación implica la creación de la variable con el valor dado. Usar una variable (identificador) no asignado previamente provocará errores semánticos en las sentencias u operaciones dónde se use.
- La asignación compuesta (Véase Section 2.1.5) usa operadores numéricos, por tanto, es necesario que la variable exista previamente, que sea de tipo numérica y que no sea constante.
- La asignación (junto a la palabra clave `const`) permite la creación de variables constantes y no se pueden reasignar.
- El tipo de la variable cambia al tipo del valor asignado.
- Se permite asignación múltiple, es decir, asignar el valor de una variable a otra.

Sintaxis

Asignación simple

```
identificador := valor, expresión, constante (de cualquier tipo)
```

Asignación encadenada o múltiple

```
identificador := identificador2 := valor, expresión, constante (de cualquier tipo)
```

Puede crecer de tamaño añadiendo mas identificadores

Asignación compuesta

```
identificador +:= valor o expresión numérica
```

(Nótese que `+:=` puede ser cambiado por otro operador compuesto (`-:=`, `*:=`, etc.) (Véase Section 2.1.5))

Asignación de constantes

```
const identificador := valor, expresión, constante o variable (de cualquier tipo)
```

2.2.2. Lectura

Permite cambiar el valor de una variable (identificador) en tiempo de ejecución del programa al valor escrito por teclado.

Existen dos versiones:

- `READ` Permite la asignación de valores numéricos.
- `READ_STRING` Permite la asignación de valores alfanuméricos (cadenas).

Características:

- Solo se admiten valores numéricos o alfanuméricos.
- Permite el cambio de tipo de la variable.
- No es necesario que la variable (identificador) exista previamente.
- No permite cambiar el valor de constantes numéricas o alfanuméricas.
- Si la cadena introducida contiene comillas simples, se eliminan. Para insertarlas sin que se eliminen se usa \'

Sintaxis

Lectura numérica

```
READ(identificador)
```

Lectura alfanumérica

```
READ_STRING(identificador)
```

2.2.3. Escritura

Permite mostrar el valor de una variable (identificador) por pantalla.

Existen dos variaciones:

Existen dos versiones:

- `PRINT` La sentencia más completa que permite todo lo explicado en características.
- `PRINT_STRING` Solo permite mostrar cadenas, variables alfanuméricas, constantes alfanuméricas y el resultado de expresiones con el operador de concatenación (`||`)

Características:

- Puede mostrar el resultado de una expresión numérica, lógica o alfanumérica con cualquier operador (aplicando las reglas de cada uno)
- Muestra el valor de constantes o variables.
- Permite la interpretación del salto de línea y tabulador.

Sintaxis

Escritura «mejorada»

```
PRINT(expresión, valor, constante, variable numérica, alfanumérica o lógica)
```

Escritura alfanumérica

```
PRINT_STRING(expresión alfanumérica)
```

2.2.4. Sentencias de control: Sentencia condicional simple y compuesta

Permite la ejecución de las sentencias si se cumple la condición. En caso negativo, se ejecutará el otro bloque (si la condicional es compuesta).

Características:

- La condición se evalúa como una expresión lógica.
- Los números o expresiones numéricos se evalúan como 1 (`true`) si son distintos de 0, y como 0 (`false`) si es 0.
- Las cadenas o expresiones alfanuméricas siempre se evalúan como 0 (`false`)

Sintaxis

Condicional simple

```
IF (condición) THEN lista de sentencias END_IF
```

Condicional compuesta

```
IF (condición) THEN lista de sentencias ELSE lista de sentencias END_IF
```

La anidación de la misma u otras sentencias en la lista de sentencias está permitida

Los espacios o nuevas líneas están permitidos entre los tokens

2.2.5. Sentencias de control: Sentencia etiqueta

Sentencia condicional formada por «labels» que según el resultado de la expresión permite ejecutar las sentencias de la «label» que contenga el mismo resultado. Se asemeja a varias sentencias `if` encadenadas pero no se evalúa cada «label».

Características:

- La «condición» y «labels» puede ser cualquier expresión, constante o variable de cualquier tipo.
- No se permite «labels» duplicadas.
- Existe una etiqueta «default» que es opcional y se ejecutará cuando el resultado de la «condición» no coincida con ninguna «label».
- El tipo de la «condición» y de todas las «label» deben ser iguales.

Sintaxis

```
CASE (condición) VALUE expresión1: lista de sentencias VALUE expresión2: lista de
        sentencias [DEFAULT: lista de sentencias] END_CASE
```

La anidación de la misma u otras sentencias en la lista de sentencias está permitida por cada value (label)

Los espacios o nuevas líneas están permitidos entre los tokens

default debe ir siempre al final pero es opcional

2.2.6. Sentencias de control: Bucle while

El bucle permite la ejecución constante de la lista de sentencias mientras se cumpla la condición.

Características:

- La condición se evalúa como una expresión lógica.
- Los números o expresiones numéricos se evalúan como 1 (`true`) si son distintos de 0, y como 0 (`false`) si es 0.
- Las cadenas o expresiones alfanuméricas siempre se evalúan como 0 (`false`)

Sintaxis

```
WHILE (condición) DO lista de sentencias END_WHILE
```

La anidación de la misma u otras sentencias en la lista de sentencias está permitida

Los espacios o nuevas líneas están permitidos entre los tokens

2.2.7. Sentencias de control: Bucle repeat-until

El bucle permite la ejecución constante de la lista de sentencias hasta que la condición sea verdadera, es decir, se ejecuta mientras sea falsa. Se garantiza siempre una ejecución como mínimo.

Características:

- La condición se evalúa como una expresión lógica.
- Los números o expresiones numéricos se evalúan como 1 (`true`) si son distintos de 0, y como 0 (`false`) si es 0.
- Las cadenas o expresiones alfanuméricas siempre se evalúan como 0 (`false`)

Sintaxis

```
REPEAT lista de sentencias UNTIL (condicion)
```

La anidación de la misma u otras sentencias en la lista de sentencias está permitida

Los espacios o nuevas líneas están permitidos entre los tokens

2.2.8. Sentencias de control: Bucle do-while

Al contrario que el bucle `repeat until`, este permite la ejecución constante de la lista de sentencias hasta que la condición sea falsa, es decir, se ejecuta mientras sea verdadera. Se garantiza siempre una ejecución como mínimo.

Características:

- La condición se evalúa como una expresión lógica.
- Los números o expresiones numéricos se evalúan como 1 (`true`) si son distintos de 0, y como 0 (`false`) si es 0.
- Las cadenas o expresiones alfanuméricas siempre se evalúan como 0 (`false`)

Sintaxis

```
DO lista de sentencias WHILE (condicion)
```

La anidación de la misma u otras sentencias en la lista de sentencias está permitida

Los espacios o nuevas líneas están permitidos entre los tokens

2.2.9. Sentencias de control: Bucle for

El bucle `for` ejecuta las instrucciones del cuerpo un número determinado de iteraciones.

Es un bucle de rango numérico, es decir, a diferencia de otros bucles que se ejecutan si una condición se cumple, este itera usando un identificador como contador que va desde un valor inicial `from` hasta un valor final `to` con un «salto» `step` determinado y parará cuando el contador sea igual al valor `to`

Características:

- El identificador cambiará (o se creará) como tipo numérico con el valor inicial de `from`
- Los valores de `from`, `to` y `step` solo pueden ser una expresión numérica, número o constante numérica.
- El `step` no es obligatorio declararlo. Su valor por defecto es 1.
- Cuando finalice el bucle, el identificador contendrá como valor final el valor de `to`. El identificador es accesible desde fuera del bucle.
- Se permite rangos negativos o positivos e iteraciones inversas, pero el intervalo de valores debe estar bien definido.

Sintaxis

```
FOR identificador FROM expresión numérica TO expresión numérica STEP expresión
                    numérica DO lista de sentencias END_FOR
```

La anidación de la misma u otras sentencias en la lista de sentencias está permitida

Los espacios o nuevas líneas están permitidos entre los tokens

2.2.10. Sentencia auxiliar: Block

Esta sentencia no tiene ningún significado especial. No es necesaria su uso en ningún sitio pero puede servir de ayuda en la mejora visual u organización de código.

Se puede usar para englobar cualquier sentencia (no produce cambios) siendo más util en lista de sentencias largas de bucles o condicionales.

Sintaxis

```
{ lista de sentencia(s) }
```

2.2.11. Sentencias especiales

Estas sentencias se podrían decir que no forman parte del lenguaje. Su uso y existencia se debe a que el lenguaje está basado en un intérprete (con modo interactivo) y por tanto puede ser beneficioso para los programas que se creen. Por ejemplo, si se elimarán estas sentencias el lenguaje no perdería capacidad como si se borrara la sentencia `if`

Las sentencias especiales son:

- `CLEAR_SCREEN` Permite limpiar la pantalla.
- `PLACE` Coloca el cursor en las coordenadas indicadas.
- `STYLE` El texto en pantalla será mostrado con colores hasta que indique.

Sintaxis

Limpiado de pantalla

```
CLEAR_SCREEN
```

Movimiento cursor

```
PLACE ( expresión numérica, expresión numérica )
```

Estilizado de texto

```
STYLE ( cadena )
```

3. Tabla de Símbolos

La tabla de símbolos contiene información sobre todos los tokens del programa. En nuestro caso nuestra tabla de símbolos es preinstalada con las *keywords*, funciones built-in y constantes lógicas y numéricas.

La tabla de símbolos esta relacionada todo el tiempo con el analizador léxico y sintáctico y se usa tanto para guardar información como para leerla en cualquier momento.

A continuación, mostramos la jerarquía de la las clases utilizadas para implementar la tabla de símbolos:

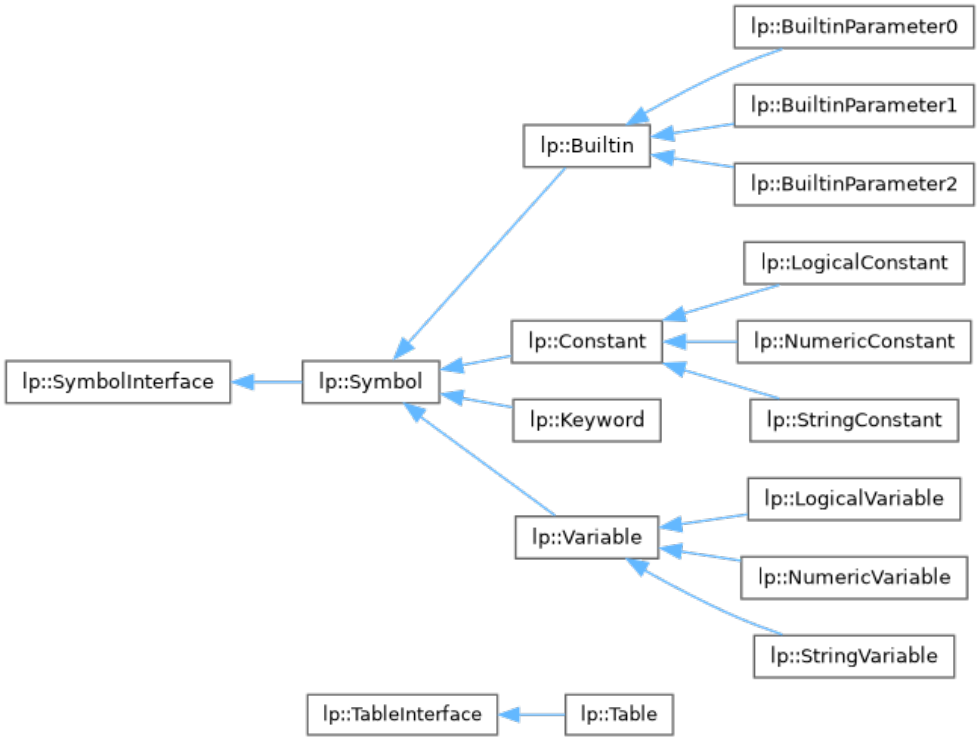

Figura 3.1: Jerarquía de clases de la tabla de símbolos

Como podemos observar de la jeraquía, la clase padre *lp::SymbolInterface* es una interfaz para las funciones que se implementarán propiamente en cada clase. De la clase hija *lp::Symbol* heredan otras 4 clases que representan a los cuatro tipos de símbolos que contendrá nuestra tabla:

- *lp::Builtin* Representa a funciones built-in.
- *lp::Constant* Representa a las constantes
- *lp::Keyword* Representa a las palabras clave
- *lp::Variable* Representa a las variables

La clase *lp::Builtin* tiene a su vez 3 clases hijas según el número de parámetros de la función que implementa métodos propios.

En el caso de las clases *lp::Constant* y *lp::Variable* sus clases hijas representan los tipos de constantes/variables: numéricas, lógicas y cadenas.

A continuación, se muestra un diagrama completo con todos métodos que podemos encontrar en cada una de las clases:

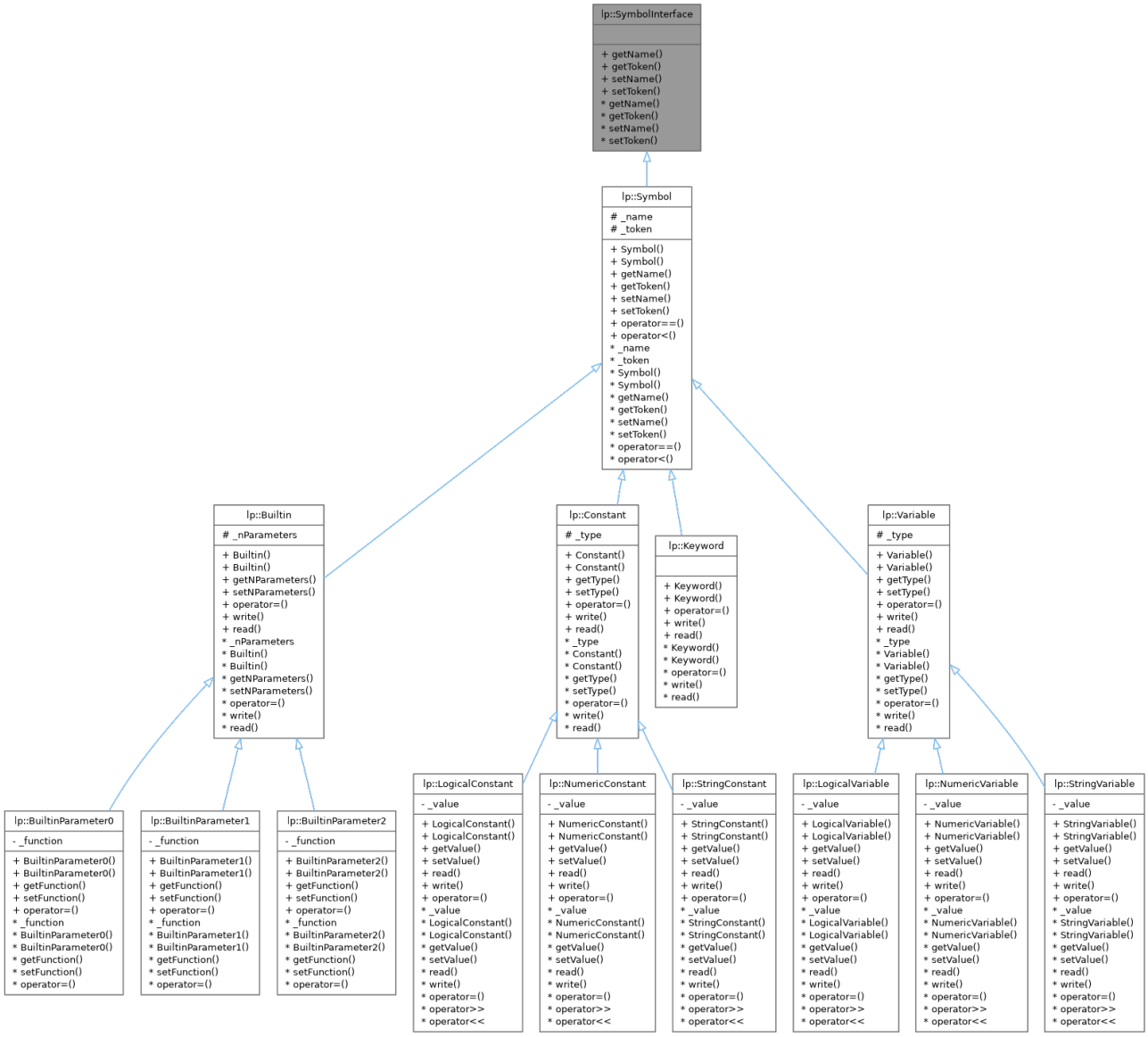

Figura 3.2: Métodos de los símbolos

La clase *lp::TableInterface* sirve como interfaz a la clase *lp::Table* que implementa los métodos necesarios para añadir, buscar o eliminar símbolos de la tabla.

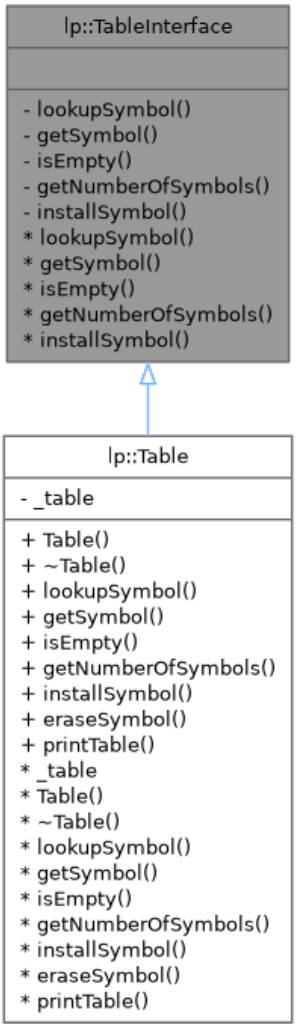

Figura 3.3: Métodos de la tabla

4. Análisis léxico

El análisis léxico es la única fase en la que se tiene contacto con el código fuente. El analizador, en este claso *Flex*, escanea cada *token* y lo envía hacia el analizador sintáctico.

Durante el análisis se accede a la tabla de símbolos para instalar nuevos símbolos (identificadores) u obtener el símbolo si ya ha sido escaneado previamente.

En esta fase pueden ocurrrir errores léxicos. En nuestro caso los errores léxicos serán comentarios de varias líneas sin abrir o terminar, cadenas sin terminar o algún otro símbolo que no coincida con ninguna de las reglas del analizador.

4.1. Definiciones regulares

A continuación, se muestran las definiciones que se han usado en el resto de las reglas del analizador:

Listing 4.1: Definiciones regulares

```
1  DIGIT  [0-9]
2
3  LETTER [a-zA-Z]
4
5  UNDERLINE [_]
6
7  NUMBER {DIGIT}+(\.?{DIGIT}*((E|e)[+-]?{DIGIT}+)?)?
8
9  NUMBER2 {DIGIT}*(\.?{DIGIT}+((E|e)[+-]?{DIGIT}+)?)?
10
11 IDENTIFIER  {LETTER}({LETTER}|{DIGIT}|{UNDERLINE}({LETTER}|{DIGIT}))*
```

Los identificadores y números se corresponde con lo explicado en la Section 2.1.1.

En el caso de los números se observa que la expresión regular engloba a número enteros, reales y de notación científica pero se divide en dos números para evitar que números del tipo `.e2` se permitan.

4.2. Comentarios y espacios

Listing 4.2: Comentarios y espacios. Analizador léxico.

```
1  [ \t]    { ; }     /* skip white space and tabular */
2
3  #.*      { ; }    /* One line comments */
4
5  <INITIAL><<+          {BEGIN(ESTADO_COMENTARIO); comment = yytext;}
6  <INITIAL>>>+     {BEGIN(INITIAL); warning("Lexical error: unopen comment", yytext);}
7  <ESTADO_COMENTARIO>>>+       BEGIN(INITIAL);
8  <ESTADO_COMENTARIO>.   // eat comment in chunks
9  <ESTADO_COMENTARIO><<EOF>> {++yynerrs; warning("Lexical error: unterminated comment", ...
       comment); yyterminate();}
10 <ESTADO_COMENTARIO>\n           lineNumber++;
```

En la línea 1 se desecha los espacios y tabuladores. En la línea 2 se «come» los comentarios de una línea y en la línea 5 comienza el autómata que reconoce el comentario de varias líneas. Como se puede observar se detecta comentarios no abiertos si se encuentra el delimitador final o comentarios no cerrados si se encuentra el fin de fichero.

4.3. Signos de puntuación

Se reconocen directamente usando los símbolos en las reglas sin necesidad de expresiones regulares:

Listing 4.3: Signos de puntuación. Analizador léxico.

```
1  ":"      {return COLON;}
2
3  ";"      {
4               /* NEW in example 5 */
5               return SEMICOLON;
6          }
7
8
9  ","      {
10              /* NEW in example 14 */
11              return COMMA;
12         }
13
14 "("              { return LPAREN; }
15 ")"              { return RPAREN; }
16
17 "{"              { return LETFCURLYBRACKET; }
18
19 "}"              { return RIGHTCURLYBRACKET; }
```

4.4. Identificadores

Listing 4.4: Identificadores. Analizador léxico.

```
1  {IDENTIFIER}          {
2                            /* NEW in example 7 */
3                            std::string identifier(yytext);
4
5                            /*Convertimos a minusculas */
6
7                            std::transform(identifier.begin(), identifier.end(), ...
                                 identifier.begin(), ::tolower);
8
9                           /*
10                            strdup() function returns a pointer to a new string
11                            which  is a duplicate of the string yytext
12                          */
13                            yylval.string =  strdup(identifier.c_str()); /* Con c_str() ...
                                  obtenemos la string a char * como en C */
14
15                          /* If the identifier is not in the table of symbols then it is ...
                               inserted */
16                           if (table.lookupSymbol(identifier) == false)
17                              {
18                                  /*
19                                   The identifier is inserted into the symbol table
20                                   as undefined Variable with value 0.0
21                                  */
22                                  lp::NumericVariable *n = new ...
                                        lp::NumericVariable(identifier,VARIABLE,UNDEFINED,0.0);
23
24                                  /* A pointer to the new NumericVariable is inserted ...
                                        into the table of symbols */
25                                  table.installSymbol(n);
26
27
28
29                                  return VARIABLE;
30                              }
31
32                          /* MODIFIED in example 11 */
33                          /*
34                              If the identifier is in the table of symbols then its token ...
                                    is returned
35                                  The identifier can be a variable or a numeric constant
36                          */
37                          else
38                              {
39                                  lp::Symbol  *s = table.getSymbol(identifier);
40
41
42                                  /* std::cout << "lex: "<< s->getName()
43                                            << "token " << s->getToken()
44                                            << std::endl; */
45
46
47                                  /*  If the identifier is in the table then its token is ...
                                        returned  */
48                                  return s->getToken();
49                              }
50                       }
```

Cuándo se detecta un identificador (véase Section 2.1.2) se transforma a minúsculas ya que nuestro lenguaje no distuingue mayúsculas y minúsculas, y por tanto se necesita instalar/comparar de la misma forma en la tabla de símbolos.

Si el símbolo no existe se instala en la tabla de símbolos como *UNDEFINED* con valor 0.0 y se devuelve el token *VARIABLE*. Si ya existe se devuelve el símbolo.

4.5. Operadores

Se reconocen directamente usando los símbolos en las reglas sin necesidad de expresiones regulares:

Listing 4.5: Operadores. Analizador léxico.

```
"--"            {return MINUSMINUS; }
"++"            {return PLUSPLUS; }

"-"             { return MINUS;   }
"+"             { return PLUS;    }

"*"             { return MULTIPLICATION; }
"/"             { return DIVISION; }
"//"            { return INTEGER_DIVISION; }


"||"            {return CONCATENATION; }


"%"             { return MODULO; }

"^"             { return POWER; }

"+:="           { return PLUS_ASSIGNMENT; }
"-:= "          { return MINUS_ASSIGNMENT; }
"*:= "          { return MULTIPLICATION_ASSIGNMENT; }
"/:= "          { return DIVISION_ASSIGNMENT; }
"//:= "         { return INTEGER_DIVISION_ASSIGNMENT; }
"%:= "          { return MODULO_ASSIGNMENT; }
"^:= "          { return POWER_ASSIGNMENT; }



":="            { return ASSIGNMENT; }


"="             { return EQUAL; }

"<>"            { return NOT_EQUAL; }

">="            { return GREATER_OR_EQUAL; }

"<="            { return LESS_OR_EQUAL; }

">"             { return GREATER_THAN; }

"<"             { return LESS_THAN; }
```

4.6. Fin de fichero

Cuando se detecta el final de fichero «EOF» el intérprete finaliza.

Listing 4.6: Fin fichero. Analizador léxico.

```
<<EOF>> { /* The interpreter finishes when finds the end of file character */
          return 0; }
```

5. Análisis sintáctico

Es la fase que sige al análisis léxico. Una vez recibido los tokens se comprueba que regla sintáctica ocurre y se ejecuta la acción semántica a ella.

A continuación, se listan los símbolos terminales y no terminales y las reglas de producción de la gramática:

5.1. Símbolos terminales

Tabla 5.1: Símbolos terminales. Análisis sintáctico.

AND	ASSIGNMENT	BUILTIN	CASE
CLEAR	COLON	COMMA	CONCATENATION
CONST	DEFAULT	DIVISION	DIVISION_ASSIGNMENT
DO	ELSE	END_CASE	END_FOR
END_IF	END_WHILE	EQUAL	FOR
FROM	GREATER_OR_EQUAL	GREATER_THAN	IF
INTEGER_DIVISION	INTEGER_DIVISION_ASSIGNMENT	LESS_OR_EQUAL	LESS_THAN
LETFCURLYBRACKET	LOGICAL_CONSTANT	LPAREN	MINUS
MINUSMINUS	MINUS_ASSIGNMENT	MODULO	MODULO_ASSIGNMENT
MULTIPLICATION	MULTIPLICATION_ASSIGNMENT	NOT	NOT_EQUAL
NUMBER	NUMERIC_CONSTANT	OR	PLUS
PLUSPLUS	PLUS_ASSIGNMENT	POWER	POWER_ASSIGNMENT
PRINT	PRINT_STRING	READ	READ_STRING
REPEAT	RIGHTCURLYBRACKET	RPAREN	SEMICOLON
STEP	STRING	STRING_CONSTANT	STYLE
THEN	TO	UNTIL	UNDEFINED
UNARY	VALUE	VARIABLE	WHILE
YYEOF			

5.2. Símbolos no terminales

Tabla 5.2: Símbolos no terminales. Análisis sintáctico.

asgn	auxasgn	auxcad	block
case	command	cond	controlSymbol
default	do_while	exp	expcad
explog	expnum	for	if
listOfExp	listOfValues	print	print_string
read	read_string	repeat	restOfListOfExp
stmt	stmtlist	while	

5.3. Reglas de producción de la gramática

Listing 5.1: Reglas de producción. Análisis sintáctico.

```
1
2   program: stmtlist
3
4   stmtlist: epsilon
5           | stmtlist stmt
6
7   stmt: SEMICOLON
8       | command SEMICOLON
9       | asgn SEMICOLON
10      | print SEMICOLON
11      | print_string SEMICOLON
12      | read SEMICOLON
13      | read_string SEMICOLON
14      | if SEMICOLON
15      | case SEMICOLON
16      | while SEMICOLON
17      | do_while SEMICOLON
18      | block
19      | repeat SEMICOLON
20      | for SEMICOLON
21      | VARIABLE PLUSPLUS
22      | PLUSPLUS VARIABLE
23      | VARIABLE MINUSMINUS
24      | MINUSMINUS VARIABLE
25
26  command: CLEAR
27         | PLACE LPAREN expnum COMMA expnum RPAREN
28         | STYLE LPAREN STRING RPAREN
29
30  block: LETFCURLYBRACKET stmtlist RIGHTCURLYBRACKET
31
32  controlSymbol: epsilon
33
34  if: IF controlSymbol cond THEN stmtlist END_IF
35    | IF controlSymbol cond THEN stmtlist ELSE stmtlist END_IF
36    | error END_IF
37
38  case: CASE controlSymbol LPAREN exp RPAREN listOfValues default END_CASE
39
40  default: epsilon
41         | DEFAULT COLON stmtlist
42
43  listOfValues: epsilon
44              | listOfValues VALUE exp COLON stmtlist
45
46  while: WHILE controlSymbol cond DO stmtlist END_WHILE
47
48  do_while: DO stmtlist WHILE controlSymbol cond
49
50  repeat: REPEAT controlSymbol stmtlist UNTIL cond
51
52  for: FOR controlSymbol VARIABLE FROM expnum TO expnum STEP expnum DO stmtlist END_FOR
53     | FOR controlSymbol VARIABLE FROM expnum TO expnum DO stmtlist END_FOR
54     | FOR NUMERIC_CONSTANT FROM expnum TO expnum DO stmtlist END_FOR
55     | FOR LOGICAL_CONSTANT FROM expnum TO expnum DO stmtlist END_FOR
56     | FOR STRING_CONSTANT FROM expnum TO expnum DO stmtlist END_FOR
57     | error END_FOR
58
59  cond: LPAREN exp RPAREN
60
61  asgn: VARIABLE ASSIGNMENT exp
```

```
62        | VARIABLE PLUS_ASSIGNMENT expnum
63        | VARIABLE MINUS_ASSIGNMENT expnum
64        | VARIABLE MULTIPLICATION_ASSIGNMENT expnum
65        | VARIABLE DIVISION_ASSIGNMENT expnum
66        | VARIABLE INTEGER_DIVISION_ASSIGNMENT expnum
67        | VARIABLE POWER_ASSIGNMENT expnum
68        | VARIABLE MODULO_ASSIGNMENT expnum
69        | CONST VARIABLE ASSIGNMENT exp
70        | VARIABLE ASSIGNMENT asgn
71        | VARIABLE PLUS_ASSIGNMENT asgn
72        | VARIABLE MINUS_ASSIGNMENT asgn
73        | VARIABLE MULTIPLICATION_ASSIGNMENT asgn
74        | VARIABLE DIVISION_ASSIGNMENT asgn
75        | VARIABLE INTEGER_DIVISION_ASSIGNMENT asgn
76        | VARIABLE POWER_ASSIGNMENT asgn
77        | VARIABLE MODULO_ASSIGNMENT asgn
78        | CONST VARIABLE ASSIGNMENT asgn
79        | VARIABLE PLUS_ASSIGNMENT auxasgn
80        | VARIABLE MINUS_ASSIGNMENT auxasgn
81        | VARIABLE MULTIPLICATION_ASSIGNMENT auxasgn
82        | VARIABLE DIVISION_ASSIGNMENT auxasgn
83        | VARIABLE INTEGER_DIVISION_ASSIGNMENT auxasgn
84        | VARIABLE POWER_ASSIGNMENT auxasgn
85        | VARIABLE MODULO_ASSIGNMENT auxasgn
86        | NUMERIC_CONSTANT ASSIGNMENT exp
87        | LOGICAL_CONSTANT ASSIGNMENT exp
88        | STRING_CONSTANT ASSIGNMENT exp
89        | NUMERIC_CONSTANT ASSIGNMENT asgn
90        | LOGICAL_CONSTANT ASSIGNMENT asgn
91        | STRING_CONSTANT ASSIGNMENT asgn
92        | CONST NUMERIC_CONSTANT ASSIGNMENT exp
93        | CONST LOGICAL_CONSTANT ASSIGNMENT exp
94        | CONST STRING_CONSTANT ASSIGNMENT exp
95        | CONST NUMERIC_CONSTANT ASSIGNMENT asgn
96        | CONST LOGICAL_CONSTANT ASSIGNMENT asgn
97        | CONST STRING_CONSTANT ASSIGNMENT asgn
98
99    auxasgn: explog
100           | expcad
101
102   print: PRINT LPAREN exp RPAREN
103        | PRINT LPAREN RPAREN SEMICOLON
104        | error SEMICOLON
105
106   print_string: PRINT_STRING LPAREN auxcad RPAREN
107
108   read: READ LPAREN VARIABLE RPAREN
109       | READ LPAREN NUMERIC_CONSTANT RPAREN
110       | READ LPAREN LOGICAL_CONSTANT RPAREN
111       | READ LPAREN STRING_CONSTANT RPAREN
112
113   read_string: READ_STRING LPAREN VARIABLE RPAREN
114              | READ_STRING LPAREN NUMERIC_CONSTANT RPAREN
115              | READ_STRING LPAREN LOGICAL_CONSTANT RPAREN
116              | READ_STRING LPAREN STRING_CONSTANT RPAREN
117
118   expnum: NUMBER
119         | VARIABLE PLUSPLUS
120         | PLUSPLUS VARIABLE
121         | VARIABLE MINUSMINUS
122         | MINUSMINUS VARIABLE
123         | expnum PLUS expnum
124         | expnum MINUS expnum
125         | expnum MULTIPLICATION expnum
126         | expnum DIVISION expnum
127         | expnum INTEGER_DIVISION expnum
128         | LPAREN expnum RPAREN
```

```
129         | PLUS expnum
130         | MINUS expnum
131         | expnum MODULO expnum
132         | expnum POWER expnum
133         | VARIABLE
134         | NUMERIC_CONSTANT
135         | BUILTIN LPAREN listOfExp RPAREN
136
137   explog: LPAREN explog RPAREN
138         | LOGICAL_CONSTANT
139         | exp GREATER_THAN exp
140         | exp GREATER_OR_EQUAL exp
141         | exp LESS_THAN exp
142         | exp LESS_OR_EQUAL exp
143         | exp EQUAL exp
144         | exp NOT_EQUAL exp
145         | exp AND exp
146         | exp OR exp
147         | NOT exp
148
149   exp: expnum
150      | explog
151      | expcad
152
153   expcad: LPAREN expcad RPAREN
154         | STRING
155         | STRING_CONSTANT
156         | auxcad CONCATENATION auxcad
157
158   auxcad: expcad
159         | VARIABLE
160
161   listOfExp: epsilon
162            | exp restOfListOfExp
163
164   restOfListOfExp: epsilon
165                  | COMMA exp restOfListOfExp
```

5.4. Acciones semánticas

De forma resumida, por cada regla sintáctica aceptada, se creará un nodo «Statement» si es una sentencia o un nodo «ExpNode» correspondiente al operador, expresión o valor. Los resultados de evaluar los «ExpNode» son usados por las sentencias como parámetros (por ejemplo condiciones). Todas las sentencias son guardadas en una lista de sentencias global y después cada sentencia es ejecutada en el modo interactivo o en el modo por fichero si no se produce ningún tipo de error.

Debido a las numerosas reglas solo se hará un resumen y se explicarán en detalle las más importantes (siempre se hace referencia al código numerado de la sección anterior):

En el primer caso, la regla de la línea 2 crea un nuevo AST y se lo asigna a la raíz (variable externa). Dependiendo de si el modo interactivo está activado o no se mostrán mensajes de que el análisis ha ido bien, número de errores, etc.

La regla de la línea 5 se encarga de rellenar el vector de la lista de sentencias y/o ejecutarlo si el modo interactivo está activado.

La mayoría de reglas crean nodos del árbol AST de su respectiva sentencia u operador. En el caso de sentencias de flujo se usa una variable «control» para controlar la ejecución del AST en el modo interactivo.

En el caso de la sentencia «case» la lista de valores es una structura con 3 campos distintos que son mapas de la STL usados para almacenar la «label» y la lista de sentencias asociada a esa «label». El uso del «map» nos permite que solo se introduzcan valores únicos, como es el caso del «case» que las «label» deben ser únicas. Además, permite que se encuentre rápidamente la label que satisface la expresión siendo más eficiente que usar un bucle for para recorrer cada label y comprobar la condición. Aunque la forma más eficaz hubiera sido una implementación usando «jump tables».

Por otro lado, la expresión y las «label» deben ser del mismo tipo, entonces para evitar insertar en el mapa valores con tipos incorrectos se recurre al siguiente truco:

Como al final el usuario debe decidir el tipo para el «case», el eligirá si usar el tipo de la expresión o la «label», por tanto, cuando se guarda la primera «label» en el mapa correspondiente este se rellena y los otros permanecen vacíos. La siguiente vez que la regla sintáctica sea validada, el mapa a usar será el que tiene tamaño distinto de 0, significando que todas las label deben ser de ese tipo. Si la label nueva añadir no coincide con el tipo del mapa no se añade y se muestra un error. Si todo va correctamente solo habrá un mapa relleno. Luego, en el nodo de la sentencia «case» se evalúa el tipo de la expresión y se comprueba si coincide con el mapa relleno.

En resumen, la primera «label» marca el tipo que deben tener las consecuentes «label». Luego en el nodo de la sentencia «case» se calcula el tipo de la expresión que debe coincidir con el tipo de mapa relleno.

Otra sentencia especial seria el incremento y decremento: Se puede usar tanto como número dando un resultado numérico como una sentencia por separada. Al fin y al cabo, el decremento o incremento es una forma abreviada de la asignación compuesta que es una sentencia pero a diferencia de otras sentencias de asignación, devuelve un valor numérico al ser evaluada. Por eso se puede encontrar el incremento o decremento como sentencia (reglas de la línea 22) o como números (reglas de la línea 120).

Por último, podemos encontrar varias reglas sintácticas de control de errores por algunas sentencias. Solo se han puesto algunas básicas y a modo de ejemplo ya que los errores que se pueden cometer son muchos y muy diferentes.

6. Código de AST

Como se ha mencionado anteriormente, el AST está compuesto por «Statement» y «ExpNode». Los «Statement» son las sentencias usadas en el programa y el objetivo final es evaluarlas para que produzcan un comportamiento en el programa. Los «ExpNode» sirven como evaluación de distintos operadores o variables, es decir, su resultado es usado por las sentecias.

Debido a la longitud del AST solo se incluirán las figuras de herencia:

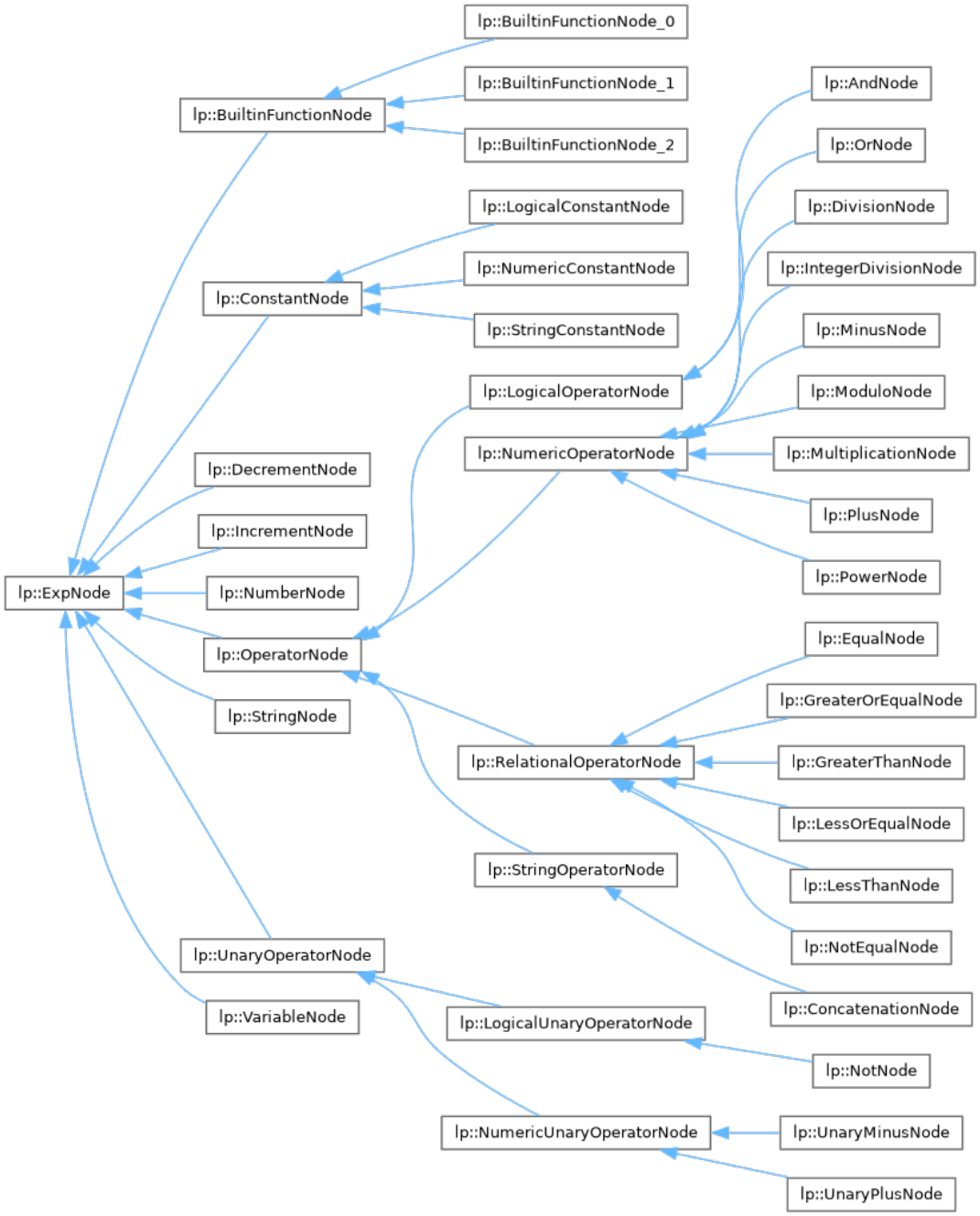

Figura 6.1: Herencia de clases ExpNode

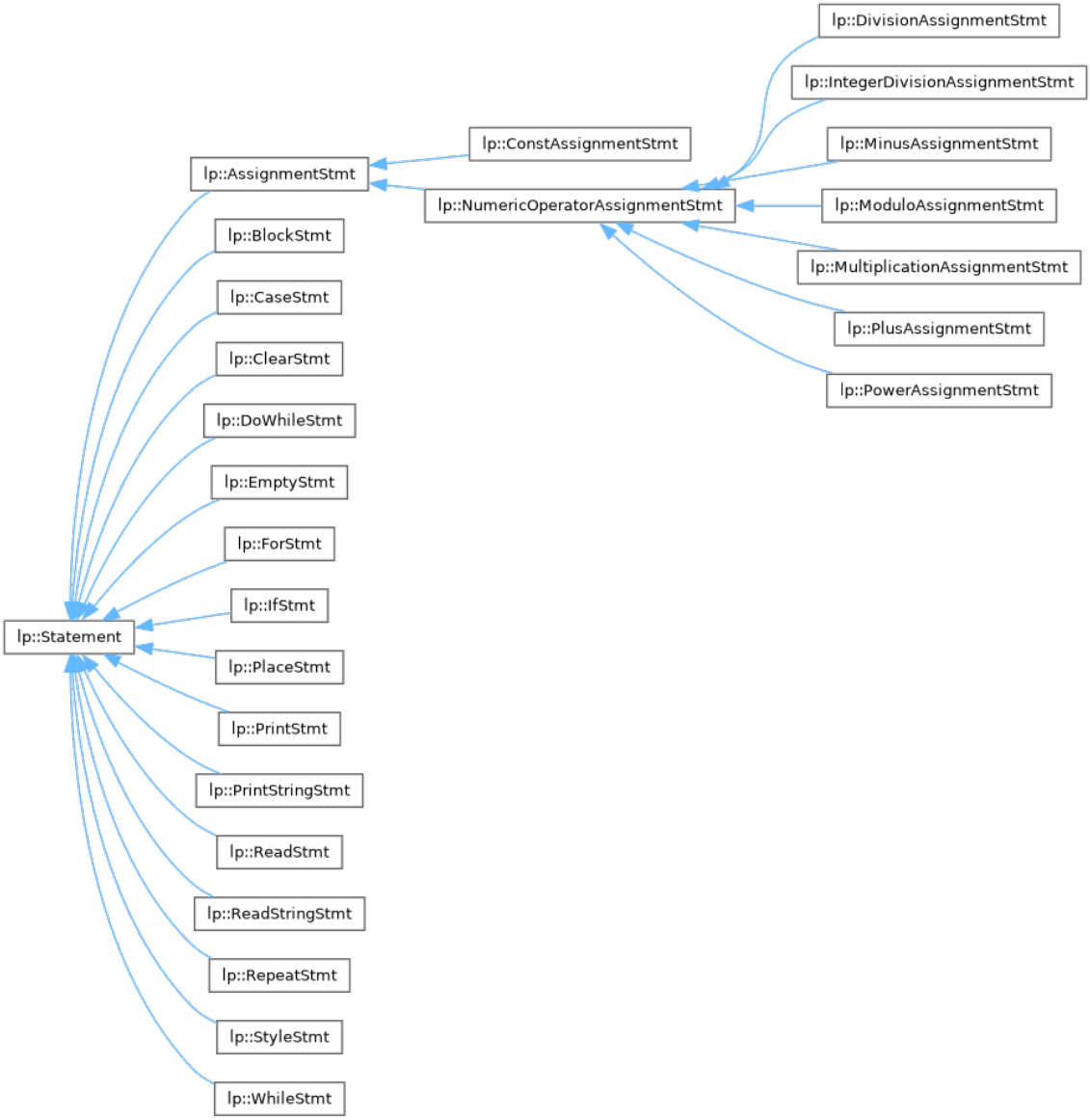

Figura 6.2: Herencia de clases Statement

De forma resumida, la clase `lp::ExpNode` es una clase abstracta dónde sus métodos se van reimplementando en las clases hijas. Las clases hijas se «subdividen» en otras según el tipo de operador o variable, y al final están las clases que representan las operaciones aritméticas con números, operadores relacionales, etc. Cada una implementa su propia lógica para evaluar las expresiones, control de errores, etc. Cada clase contiene un método para evaluar como número, lógico o cadena. Por ejemplo, `lp::PlusNode` contiene el método `evaluateNumber()` que obtiene el resultado de la suma. Algunas clases (como la `lp::NumberNode`) contiene el método `evaluateBool` ya que es usado para obtenener el número como verdaddero o falso para las condiciones.

La clase padre `lp::Statement` tiene como clases hijas todas las sentencias del lenguaje. Cada clave tiene la función principal de `printAST` y la más importante `evaluate()` que permite evaluar la sentencia. Cabe destacar que la sentencia de asignación tiene dos clases hijas, una la la asignación de constantes y otra para la asignación compuesta.

7. Funciones auxiliares

Se han implementado las funciones:

- `to_dregrees` Conversión de radianes a grados.
- `to_radians` Conversión de grados a radianes.

8. Modo de obtención del intérprete

Carpeta	Archivo	Descripción
ast	ast.cpp	Código de funciones de la clase AST
	ast.hpp	Declaración de la clase AST
error	error.cpp	Código de funciones de recuperación de errores
	error.hpp	Prototipos de funciones de recuperación de errores
includes	macros.hpp	Macros para la pantalla
parser	interpreter.l	Archivo léxico o de análisis
	interpreter.y	Archivo de gramática
table	builtin.cpp	Código de funciones de la clase Builtin
	builtin.hpp	Declaración de la clase Builtin (función incorporada)
	builtinFunction.cpp	Código de funciones built-in
	builtinFunction.hpp	Prototipos de funciones built-in
	builtinParameter0.cpp	Código de funciones de la clase BuiltinParameter0
	builtinParameter0.hpp	Declaración de la clase BuiltinParameter0
	builtinParameter1.cpp	Código de funciones de la clase BuiltinParameter1
	builtinParameter1.hpp	Declaración de la clase BuiltinParameter1
	builtinParameter2.cpp	Código de funciones de la clase BuiltinParameter2
	builtinParameter2.hpp	Declaración de la clase BuiltinParameter2
	constant.cpp	Código de funciones de la clase Constant
	constant.hpp	Declaración de la clase Constant
	init.cpp	Código de la inicialización de la tabla de símbolos
	init.hpp	Prototipo de la inicialización de la tabla de símbolos
	keyword.cpp	Código de funciones de la clase Keyword
	keyword.hpp	Declaración de la clase Keyword
	logicalConstant.cpp	Código de funciones de la clase LogicalConstant
	logicalConstant.hpp	Declaración de la clase LogicalConstant
	logicalVariable.cpp	Código de funciones de la clase LogicalVariable
	logicalVariable.hpp	Declaración de la clase LogicalVariable
	numericConstant.cpp	Código de funciones de la clase NumericConstant
	numericConstant.hpp	Declaración de la clase NumericConstant
	numericVariable.cpp	Código de funciones de la clase NumericVariable
	numericVariable.hpp	Declaración de la clase NumericVariable

Carpeta	Archivo	Descripción
table	stringConstant.cpp	Código de algunas funciones de la clase StringConstant
	stringConstant.hpp	Declaración de la clase StringVariable
	stringVariable.cpp	Código de algunas funciones de la clase LogicalVariable
	stringVariable.hpp	Declaración de la clase StringVariable
	symbol.cpp	Código de algunas funciones de la clase Symbol
	symbol.hpp	Declaración de la clase Symbol
	symbolInterface.hpp	Declaración de abstract SymbolInterface class
	table.cpp	Código de algunas funciones de la clase Table
	table.hpp	Declaración de TableInterface class
	tableInterface.hpp	Declaración de abstract TableInterface class
	variable.cpp	Código de algunas funciones de la clase Variable
	variable.hpp	Declaración de la clase Variable
/	interpreter.cpp	Programa principal

9. Modo de ejecución del intérprete

El intérprete una vez compilado usando *make* sobre el directorio raíz (puede saltar un «warning» si se compila con una versión de `g++` antigua) se obtendrá el ejecutable *interpreter.exe* en la raíz de la carpeta.

Para ejecutarlo se puede hacer de dos formas:

- **Modo interactivo:** Desde una terminal se ejecuta el ejecutable y ya estará listo para recibir sentencias.
- **Modo fichero:** Desde una terminal se ejecuta el ejecutable pasándole como argumento un archivo de texto con extensión .p

10. Ejemplos

En la carpeta */examples* hay disponibles distintos ejemplos de todas las sentencias del lenguaje. Así como programas de prueba aplicando distintas sentencias

11. Conclusión

El desarrollo del intérprete ha sido largo y con algunos problemas en como implementar algunas sentencias como el «case», errores de bison, etc. El intérprete tiene un funcionamiento sólido aunque se podría mejorar algunos aspectos en mensajes y control de errores más personalizados.

Casi todas las sentencias deben funcionar como estuvieron pensadas en algún momento (aunque el incremento y decremento no funcionan correctamente como sentencia). Un aspecto débil del intérprete es el modo interactivo: Muchas veces cuándo ejecutas algo y hay un error sintáctico, el programa se queda «trabado» esperando los tokens correctos hasta que ya empieza a analizar sentencias de nuevo correctamente. Se podría haber solucionado mejorando el control de errores con reglas sintácticas.

El punto positivo es que contiene bastantes sentencias y operandos con los que hacer diferentes programas muy diversos y se puede mejorar en un futuro con más funcionalidades.

Un punto negativo que traía de serie es que las funciones built-in son muy difíciles de implementar ya que necesita hacer muchos cambios según el tipo del valor devuelto y de los argumentos.

En conclusión, es un intérprete sólido que se puede mejorar en algunos aspectos concretos.

12. Bibliografía

https://web.iitd.ac.in/ sumeet/flex__bison.pdf

SKB. «checking unfinished comments in flex». Stack Overflow, 24 de junio de 2016, https://stackoverflow.com/q/29991073.

Lexical Analysis With Flex, for Flex 2.6.2: EOF. http://westes.github.io/flex/manual/EOF.html. Accedido 30 de junio de 2023.

IBM Documentation. 24 de marzo de 2023, https://www.ibm.com/docs/en/aix/7.1?topic=informa yacc-program-error-handling.

Prodanov, Ivan. «Is "else if" faster than "switch case"?» Stack Overflow, 2 de febrero de 2020, https://stackoverflow.com/q/767821.

Speed Test: Switch vs If-Else-If. http://www.blackwasp.co.uk/SpeedTestIfElseSwitch.aspx. Accedido 30 de junio de 2023.

https://cplusplus.com/reference/map/map/. Accedido 30 de junio de 2023.

https://www.geeksforgeeks.org/comparing-string-objects-using-relational-operators-c/

https://cplusplus.com/reference/string/string/operators/. Accedido 30 de junio de 2023.