Universidad de Córdoba

Escuela Politécnica Superior de Córdoba

Grado de Ingeniería Informática

Especialidad de Computación

Tercer Curso, segundo cuatrimestre

Curso Académico: 2018 - 2019

Procesadores de Lenguajes

Intérprete de pseudocódigo en Español

Daniel Ranchal Parrado

i62rapad@uco.es

28 de mayo de 2019, Córdoba

Índice

Índice de figuras

1. Introducción

Este trabajo consiste en la elaboración de un intérprete de pseudocódigo en español. Un lenguaje parecido al desarrollado en este trabajo es Python, el cual es un lenguaje interpretado y en el que muchas de sus sentencias son parecidas al pseudocódigo.

El lenguaje en el que está implementado este intérprete es C++. Para ello, se ha utilizado las librerías flex y bison, las cuales están implementadas en C++. La herramienta flex se encarga de reconocer los diferentes componentes léxicos que se encuentran en un fichero utilizando expresiones regulares. Mientras que la librería bison se encargará de hacer una acción u otra según las reglas sintácticas que el desarrollador escriba.

En este documento se hablará sobre los diferentes elementos que tiene el lenguaje de pseudocódigo, es decir, las diferencias sentencias se pueden utilizar. Además de esto, se comentará como se organiza la tabla de símbolos, es decir, su estructura de datos y por cuál símbolos está formada. Luego se describirá los componentes léxicos y las expresiones regulares utilizadas durante la fase del análisis léxico. Después se describirá la fase de análisis sintáctico, en el que principalmente se hablará sobre los símbolos de la gramática, las reglas de producción de la gramática y las acciones semánticas. Una vez explicado las fases léxica y sintáctica, se describirá el árbol de sintáxis abstracto explicando una a una las clases definidas y utilizadas. Además de esto, se hará un resumen sobre las funciones auxiliares desarrollada.

En las últimas secciones de este documento, se explicarán los ficheros utilizados para obtener el ejecutable del intérprete (por lo tanto se hará una descripción del fichero makefile), los diferentes modo de ejecución de este intérprete y algunos ejemplos para comprobar que el intérprete funciona correctamente.

Como último, se harán unas reflexiones sobre este trabajo y se describirán los puntos fuertes y débiles del intérprete desarrollado.

2. Lenguaje de Pseudocódigo

Como se ha comentado anteriormente en la introducción, en esta sección se describirán los diferentes componentes léxicos y las sentencias. Aquellos componentes léxicos y sentencias que no aparezcan en la documentación y se hayan incluido se escribirán en itálica, además de avisar en el texto de explicación que se ha incluido.

2.1. Componentes léxicos

- **_mod**: Este componente léxico define la operación módulo.
- **_div**: Define la operación división entera.
- **_o**: Define la operación lógica OR.
- **_y**: Define la operación lógica AND.
- **_no**: Define la operación lógica NOT.
- **leer**: Se encarga de leer por pantalla aquellas variables que sean números o booleanos.
- **leer_cadena**: Responsable de introducir en una variable de tipo string una cadena.
- **escribir**: Se encarga de mostrar por pantalla expresiones de tipo número y booleanas.
- **escribir_cadena**: Responsable de mostrar por pantalla expresiones de tipo string.
- **si**: Componente que define el principio de una sentencia if.
- **entonces**: Componente que define el comienzo de las sentencias del consecuente de if.
- **si_no**: Componente que determina el comienzo de la alternancia de la sentencia if si no se cumple la condición.

- **fin_si**: Componente léxico que determina el fin de una sentencia if.
- **mientras**: Componente léxico que define el comienzo de la sentencia while.
- **hacer**: Componente que determina el comienzo de las sentencias del bucle while.
- **fin_mientras**: Determina el final del bucle while.
- **repetir**: Componente léxico que marca el comienzo de la sentencia do-while.
- **hasta**: Componente léxico que se utiliza en las sentencias do-while y for.
- **para**: Componente léxico que determina el comienzo del bucle for.
- **fin_para**: Determina el final de la sentencia del bucle for.
- **desde**: Determina que valor inicial tomará la variable en el bucle for.
- **paso**: Determina en cuantas unidades aumenta la variable en el bucle for.
- **_borrar**: Esta palabra reservada se encarga de borrar la pantalla.
- **_lugar**: Esta palabra reservada es la encargada de mover el cursor en una posición de la pantalla.
- ***_terminar***: **¡Este componente léxico es una novedad!** Esta palabra reservada es la encargada de terminar la ejecución de un programa.
- ***_ejecutar***: **¡Este componente léxico es una novedad!** Esta palabra reservada es la encargada de ejecutar un archivo con extensión .e dentro de un programa.
- ***segun***: **¡Este componente léxico es una novedad!** Determina el comienzo de la sentencia switch.
- ***fin_segun***: **¡Este componente léxico es una novedad!** Determina el final de la sentencia switch.

- ***valor***: **¡Este componente léxico es una novedad!** Determina uno de los casos de la sentencia switch.
- ***defecto***: **¡Este componente léxico es una novedad!** Determina el comienzo del caso por defecto si ninguno de los casos se hubiera ejecutado.
- ***romper***: **¡Este componente léxico es una novedad!** Determina que solo tiene que ejecutar ese caso y no ejecutar los posteriores.
- **Identificadores**
  - Están compuestos por una serie de letras, dígitos y el subrayado.
  - Deben comenzar por una letra
  - No podrán acabar con el símbolo de subrayado, ni tener dos subrayados seguidos.
- **Número**
  - Se utilizan números enteros, reales de punto fijo y reales con notación científica.
  - Todo ellos serán tratados conjuntamente como números.
- **Cadena**
  - Están compuestos por una serie de caracteres limitados por comillas simples.
  - Permite la inclusión de la comilla simple incluyendo la barra (\).
  - Las comillas exteriores no se almacenan como parte de la cadena.
- **Operadores de asignación**
  - Asignación :=
  - *Suma Variable +:=*: **¡Este componente léxico es una novedad!**
  - *Restar Variable -:=*: **¡Este componente léxico es una novedad!**

- *Multiplicar Variable *:=*: **¡Este componente léxico es una novedad!**
- *Multiplicar Variable /:=*: **¡Este componente léxico es una novedad!**

- **Operadores aritméticos**
  - Suma + : Operador unario y binario
  - Resta - : Operador unario y binario
  - Producto *
  - División /
  - División entera _div
  - Módulo _mod
  - Potencia **
  - *Operador incremento ++* : **¡Este componente léxico es una novedad!**
  - *Operador decremento –* : **¡Este componente léxico es una novedad!**

- **Operadores alfanuméricos**
  - Concatenación ||

- **Operadores relacionales**:
  - Menor que <
  - Menor o igual que <=
  - Mayor que >
  - Mayor o igual que >=
  - *Igual que* = : **¡Este componente léxico es una novedad!** Ahora se permite comprobar booleanos.

- *Distinto que* <> : **¡Este componente léxico es una novedad!** Ahora se permite comprobar booleanos.

- **Operadores lógicos**
  - Disyunción lógica _o
  - Conjunción lógica _y

2.2. Sentencias

A continuación se procede a explicar las distintas sentencias disponibles en este lenguaje.

- **Sentencias de asignación**
  - Asignación de expresiones numéricas
  - Asignación de expresiones alfanuméricas
  - *Asignación de expresiones booleanas*: **¡Esta sentencia es una novedad!**
  - *Asignación y suma de una variable numérica +:=* : **¡Esta sentencia es una novedad!**
  - *Asignación y resta de una variable numérica -:=* : **¡Esta sentencia es una novedad!**
  - *Asignación y multiplicación de una variable numérica *:=* : **¡Esta sentencia es una novedad!**
  - *Asignación y división de una variable numérica /:=* : **¡Esta sentencia es una novedad!**

- **Sentencia de lectura**
  - Leer: Declara a identificador como variable numérica
  - Leer_cadena: Declara a identificador como variable alfanumérica

- **Sentencia de escritura**
  - Escribir: El valor de la expresión numérica es escrito en pantalla
  - Escribir_cadena: La cadena es escrita por la pantalla
- **Sentencias de control**
  - Sentencia condicional simple
  - Sentencia condicional compuesta
  - Bucle mientras
  - Bucle repetir
  - Bucle para
  - *Bucle según*: **¡Esta sentencia es una novedad!**
- **Comandos especiales**
  - _borrar
  - _lugar
  - *_terminar*: **¡Este comando es una novedad!**
  - *_ejecutar*: **¡Este comando es una novedad!**

3. Tabla de Símbolos

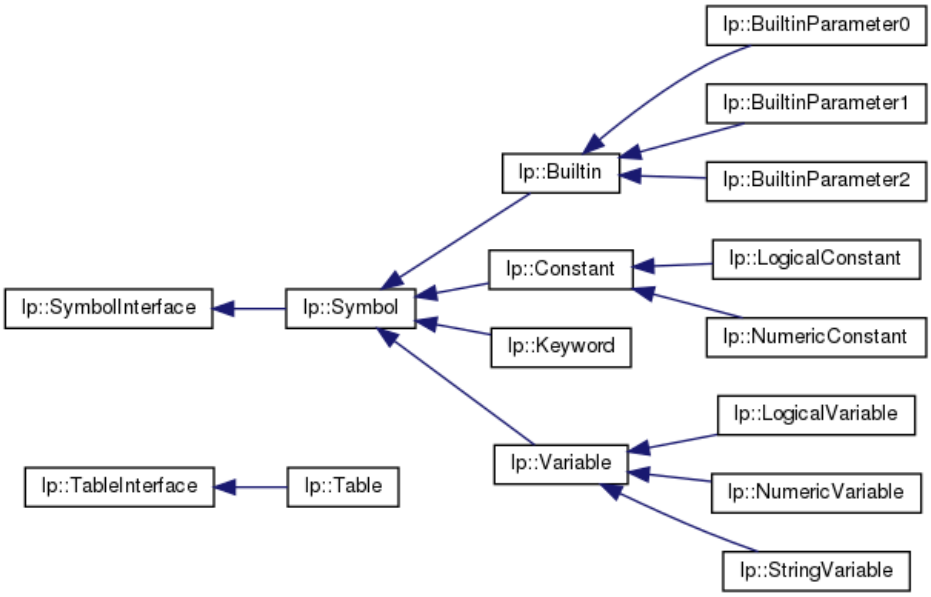

Figura 1: Jerarquía de las clases de la tabla de símbolos

En la figura 1 se puede observar la jerarquía entre las distintas clases de la tabla de símbolos. Para definir esta tabla, se ha hecho uso de dos clases virtuales, en este caso *TableInterface* y *SymbolInterface*. A partir de la primera se ha definido la clase *Table*, la cual ha heredado públicamente los métodos virtuales de la clasa abstracta. Esta clase define una estructura de datos de tipo mapa con una clave *string* y otra clave con un puntero a un objeto de la clase *Symbol*. Los principales métodos de esta clase son obtener, instalar y borrar símbolos de la estructura de datos.

SymbolInterface define los métodos virtuales para modificar y obtener el token y el nombre de un símbolo. La clase *Symbol* hereda públicamente este interfaz, y además de redefinir los métodos virtuales, define un constructor para poder crear un objeto de esta clase. La clase *Builtin* hereda públicamente los métodos de la clase *Symbol* y define un atributo, el cual define el número de

parámetros, además de los correspondientes getters y setters y su correspondiente constructor. Las clases *builtinParameter0*, *builtinParameter1* y *builtinParameter2* heredan públicamente los métodos y atributos de la clase *Builtin*. Estas clases se encargan de representar funciones que representan funciones con 0, 1 o 2 parámetros respectivamente.

La clase *Constant* hereda públicamente los métodos y atributos de la clase *Symbol*. Define el atributo _type, que determina el tipo de datos de la constante, además de los correspondientes getters y setters y un constructor. Esta clase representa un dato constante, es decir, no se puede modificar. La clase *LogicalConstant* y *numericConstant* heredan públicamente los métodos y atributos de la clase *Constant*. El primero de ellos representa una constante booleana mientras que la segunda representa una constante numérica.

La clase *Keyword* hereda públicamente los método y atributos de la clase *Symbol*. Esta clase representa las palabras clave del pseudocódigo.

La clase *Variable* hereda públicamente los método y atributos de la clase *Symbol*. Esta clase representa una variable de cualquier tipo de dato. Las clases *LogicalVariable*, *NumericVariable* y *StringVariable* heredan públicamente los métodos y atributos de la clase *Variable*. Representan variables lógicas, numéricas y strings respectivamente.

Los archivos que se encargan de inicializar la tabla son *init.cpp* y*init.hpp*, que se explicarán posteriormente.

4. Análisis léxico

```
digit     [0-9]
letter    [a-zA-Z]

identifier          {letter}({letter}|{digit}|\_({letter}|{digit}))*
number              {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
strings             "'"([^']|"\\'")*"'"
```

Tal y como se puede ver en este parte del código, se han definido las expresiones regulares de un identificador, un número y una string. Un identificador debe de empezar por un letra y luego puede tener cero o muchas letras, dígitos o una barra baja con letras y dígitos. La expresión regular de un número permite la representación de este en su forma normal o en representación científica. Mientras que la expresión regular de la string esta formada por todo aquello que no sea un ' o que sea \'. Todo esto estaría englobado por unas comillas simples.

```
%x COMMENTARY_1
%x COMMENTARY_2
%x ERROR
```

Además se han creado tres estados, los dos primeros para reconocer los dos tipos de comentarios y el tercero para reconocer estados de error.

```
"#"                             {BEGIN(COMMENTARY_1);}
<COMMENTARY_1>[^"#"|^\n]        {;}
<COMMENTARY_1>\^                {;}
<COMMENTARY_1>\n                {lineNumber++;}
<COMMENTARY_1>"#"               {BEGIN(INITIAL);}

"@"                     {BEGIN(COMMENTARY_2);}
<COMMENTARY_2>.         {;}
<COMMENTARY_2>\n        {BEGIN(INITIAL); lineNumber++;}
```

El estado del COMMENTARY_1 se activará cuando se encuentre el símbolo de la almohadilla #. Cualquier carácter que vaya a continuación será ignorado

por el léxico. Cuando se encuentre un salto de línea, se incrementará el contador del número de líneas. Una vez que se encuentre de nuevo una almohadilla #, se terminará el comentario.

Respecto al comentario de línea, cuando se encuentra una @, se activará el estado COMMENTARY_2. Cualquier carácter que vaya a continuación de este, serán ignorados por el léxico. Cuando se encuentre un salto de línea, este estado terminará y se incrementará el contador del número de líneas.

```
(?i:_o)          {return OR;}
(?i:_y)          {return AND;}
(?i:_no)         {return NOT;}
```

En este código se reconoce los operadores lógicos OR, AND y NOT. Para reconocer el operador OR, la expresión regular es (?i:_o), es decir, ignora si está en mayúscula o en minúscula. Lo mismo pasa con el resto de operadores.

```
"<"              {return LESS_THAN;}
"<="             {return LESS_OR_EQUAL;}
">"              {return GREATER_THAN;}
">="             {return GREATER_OR_EQUAL;}
"="              {return EQUAL;}
"<>"             {return NOT_EQUAL;}
```

Aquí se definen las expresiones regulares de los operadores relacionales.

```
"||"                    {return CONCATENATE;}
"**"                    {return POWER_OF;}
(?i:_mod)               {return MODULUS;}
(?i:_div)               {return INTEGER_DIVISION;}
"/"                     {return DIVISION;}
"*"                     {return PRODUCT;}
"-"                     {return MINUS;}
"--"                    {return MINUS_MINUS;}
"+"                     {return PLUS;}
"++"                    {return PLUS_PLUS;}
":="                    {return ASSIGNMENT;}
"+:="                   {return SUM_VARIABLE;}
"-:="                   {return SUBSTRACT_VARIABLE;}
"*:="                   {return PRODUCT_VARIABLE;}
"/:="                   {return DIVIDE_VARIABLE;}
```

A continuación se definen las expresiones regulares de los operadores numéricos y alfanuméricos. Estas expresiones son muy simples ya que solo buscamos el símbolo literalmente.

```
";"                     {return SEMICOLON;}
","                     {return COMMA;}
"("                     {return LEFTPARENTHESIS;}
")"                     {return RIGHTPARENTHESIS;}
":"                     {return COLON;}
```

Estas expresiones regulares representan los diferentes signos de puntuación que se encuentran en este pseudocódigo.

```
(?i:_borrar)            {return ERASE;}
(?i:_lugar)             {return PLACE;}
(?i:_terminar)          {return KILL;}
(?i:_ejecutar)          {return EXECUTE;}
```

Estas expresiones regulares representan los comandos especiales del pseudocódigo. Son insensibles a mayúsculas y a minúsculas.

```
[ \t]                   {;}
\n                      {lineNumber++;}
```

Cada vez que se encuentre un espacio o una tabulación, estos serán ignorados por el léxico. Cuando se encuentre un salto de línea, se incrementará el contador del número de líneas.

```
{identifier}{
    for(int i=0; yytext[i] != '\0'; i++){
        yytext[i]=tolower(yytext[i]);
    }

    std::string identifier(yytext);
    yylval.identifier = strdup(yytext);

    if(! table.lookupSymbol(identifier)){
        lp::NumericVariable *var = new lp::NumericVariable(
    identifier, VARIABLE, NUMBER, 0.0);
        table.installSymbol(var);
        return VARIABLE;
    }
    else{
        lp::Symbol *s = table.getSymbol(identifier);
        return s->getToken();
    }
}
```

En este trozo de código se define las acciones que se tienen que realizar cuando se encuentra un identificador. Lo primero de todo será pasar todas las letras a minúscula y pasar yytext a un std::string. A continuación le pasaremos el valor de este string al sintáctico a través de yylval.identifier.

Si el identificador no existe en la tabla de símbolos, será un nuevo identificador y se definirá como una variable numérica con valor 0. En el caso que ya exista en la tabla de símbolos, se recogerá de esta tabla y devolverá su token.

```
{number}            {
                        yylval.number = atof(yytext);
                        return NUMBER;
                    }

{strings}           {
                        std::string aux(yytext);
                        aux = aux.substr(1, aux.size()-2);
                        yylval.strings = strdup(aux.c_str());

                        return STRINGS;
                    }

<<EOF>>   {std::cout << std::endl; return 0;}
```

En el caso que se reconozca un número, se le pasará el valor del número al sintáctico a través de la variable yylval.number mientras que si se reconoce una string, se le harán las modificaciones oportunas para quitar las comillas simples que le engloban y se le pasa su valor al sintáctico a través de la varibale yylval.strings. Cuando el léxico se encuentre un fin de fichero, se terminará el programa.

```
.           {
                BEGIN(ERROR);
                yymore();
            }

<ERROR>[^0-9+\-*/()\^%\|\: \t\n\;\,\#\@a-zA-Z=<>!&]     {
                                                            yymore();
                                                        }

<ERROR>(.|\n)   {
                    yyless(yyleng-1);
                    warning("Lexical error", yytext);
                    BEGIN(INITIAL);
                }
```

Por último, si no se ha reconocido ninguna de las expresiones regulares anteriores, es que se ha introducido un carácter que no puede entender el pseudocódigo. Cuando pase esto, se activará el estado del error. Se terminará este estado cuando se encuentre un salto de línea u otro carácter no entendible.

5. Análisis sintáctico

5.1. Símbolos terminales

Como se puede ver, estos son los símbolos terminales de la gramática. Aquellos símbolos que se declaran antes tienes menos precedencia que los que se definen al final. En este caso los símbolos SEMICOLON y COLON tendrán menos precedencia que POWER_OF. Para determinar la asociatividad de las operaciones se utiliza la palabra clave %left o %right. Aquellos símbolos que vayan acompañados por %left, serán asociativos por la izquierda, por ejemplo DIVISION y los que vayan acompañados por %right serán asociativos por la derecha. Para determinar que los símbolos no son asociativos, irán acompañados de la palabra clave %nonassoc. Para determinar el tipo de dato de cualquiera de los símbolos terminales, estos irán acompañados de $< tipo\_de\_dato >$.

```
%token SEMICOLON COLON
%token KILL ERASE PLACE REPETITION UNTIL FOR FROM STEP DO END_FOR
    WHILE END_WHILE IF THEN ELSE END_IF READ READ_STRING WRITE
    WRITE_STRING SWITCH END_SWITCH VALUE BREAK DEFAULT EXECUTE
%right ASSIGNMENT
%token COMMA
%token <number> NUMBER
%token <logic> BOOL
%token <strings> STRINGS
%token <identifier> VARIABLE CONSTANT BUILTIN
%left OR
%left AND
%nonassoc LESS_THAN LESS_OR_EQUAL GREATER_THAN GREATER_OR_EQUAL
    EQUAL NOT_EQUAL
%left NOT
```

```
%left CONCATENATE
%left SUM_VARIABLE SUBSTRACT_VARIABLE PRODUCT_VARIABLE
    DIVIDE_VARIABLE
%left PLUS MINUS PLUS_PLUS MINUS_MINUS
%left PRODUCT DIVISION MODULUS INTEGER_DIVISION
%left LEFTPARENTHESIS RIGHTPARENTHESIS
%nonassoc UNARY
%right POWER_OF
```

5.2. Símbolos no terminales

Esta es la lista de símbolos no terminales que se encuentran en la gramática de este pseudocódigo. Los símbolos no terminales exp y cond son de tipo expNode, siendo esta variable un puntero a un objeto de la clase lp::ExpNode, que define una expresión numérica. Los símbolos no terminales listOfExp y restOfListOfExp son de tipo parameters, siendo parameters una lista de punteros a objetos de la clase lp::ExpNode. El símbolo no terminal stmtlist es de tipo stmts, que es una lista de punteros a objetos de la clase lp::Statement.El símbolo no terminal blocks, que se ha utilizado para implementar la sentencia de control segun, es de tipo cases, siendo esta variable una lista de punteros a objetos de la clase lp::CasesStmt, los cuales son utilizados por la sentencia de control segun para definir los distintos casos que hubiese. En la quinta línea de este trozo de código se encuentran los diferentes símbolos no terminales de tipo st, siendo esta variable un puntero a un objeto de la clase lp::Statement. El último símbolo terminal es program, que es de tipo prog, siendo esta variable un puntero a un objeto de la clase lp::AST.

```
%type <expNode> exp cond
%type <parameters> listOfExp restOfListOfExp
%type <stmts> stmtlist
%type <cases> blocks
%type <st> stmt asgn print read si mientras para repetir borrar
    lugar segun ejecutar terminar
%type <prog> program
```

5.3. Reglas de producción y acciones semánticas

Para poder explicar las distintas reglas de producción, primero hay que comentar que el símbolo inicial de esta gramática es program. Para determinar el símbolo inicial, este irá acompañado por la palabra especial **%start**.

```
program: stmtlist
        {
          $$ = new lp::AST($1);
          root = $$;
        }
;
```

Esta sería la primera regla que se ejecutaría en este gramática. Cuando se dispara, se crea un objeto de la clase AST, es decir, se crea el árbol de sintaxis abstracta.

En esta regla de producción, la primera vez que entra, se crea una lista de punteros a objetos de la clase lp::Statement. En el caso que este símbolo no terminal acompañe al símbolo no terminal stmt, este último se insertará en la lista de sentencias. En el caso que estemos en el modo interactivo, estas sentencias se ejecutarían al instante, pero este concepto se explicará en la sección 9. En el caso que se produzca un error, se disparará la tercera regla, haciendo que se restaure el símbolo look-ahead previo al error.

```
stmtlist:
        {
          $$ = new std::list<lp::Statement *>();
        }

        | stmtlist stmt
        {
          $$ = $1;
          $$->push_back($2);

          if(interactiveMode){
            if(manageInteractiveMode == 0){
              $2->evaluate();
```

```
            std::cout << std::endl << BIYELLOW << " > " << RESET;
          }
        }
      }

      | stmtlist error
      {
        $$ = $1;

        yyclearin;
      }
;
```

El símbolo no terminal stmt tiene las siguientes reglas. En el caso que solo haya un ; como sentencia, se creará una sentencia vacía. En el resto de casos se disparará una regla u otra dependiendo de la sentencia que haya. Al dispararse, se le pasa al símbolo no terminal de esa sentencia.

```
stmt: SEMICOLON
    {
      $$ = new lp::EmptyStmt();
    }

    | asgn SEMICOLON
    {
      $$ = $1;
    }

    | para SEMICOLON
    {
      $$ = $1;
    }

    | repetir SEMICOLON
    {
      $$ = $1;
    }
```

```
| mientras SEMICOLON
{
  $$ = $1;
}

| si SEMICOLON
{
  $$ = $1;
}

| print SEMICOLON
{
  $$ = $1;
}

| read SEMICOLON
{
  $$ = $1;
}

| borrar SEMICOLON
{
  $$ = $1;
}

| lugar SEMICOLON
{
  $$ = $1;
}

| segun SEMICOLON
{
  $$ = $1;
}

| ejecutar SEMICOLON
{
  $$ = $1;
```

```
    }

    | terminar SEMICOLON
    {
      $$ = $1;
    }
;
```

Esta regla determina las distintas expresiones que puede haber entre las variables. En la primera regla, el símbolo no terminal exp se sustituye por el símbolo terminal NUMBER creando un nodo de un número en el árbol de sintaxis abstracta. En la segunda regla, al igual que en la primera, el símbolo no terminal exp se sustituye por el terminal STRINGS creando un nodo de string. Lo mismo pasa con las reglas en las que se sutituye por exp por BOOL, VARIABLE y CONSTANT. En la regla que exp se sustituye por (exp), lo único que hay que hacer es que $$ siga siendo exp, por eso se ha realizado la sentencia $ = $2.

En el caso que exp se vaya a sustituir por + exp o - exp, se creará un nodo unario de + o -. Para determinar esto, en la regla hay que poner la palabra especial **%prec**.

Desde la novena hasta la vigésima cuarta regla, el símbolo no terminal exp se sustituye por el mismo, un símbolo terminal y el mismo de nuevo. Este símbolo terminal es la suma, resta, multiplicación, división, división entera, módulo, potencia, concatenación, >, >=, <, <=, =, <>, la operación AND y OR. En la última regla se reconocen las funciones que tienen 0, 1 o 2 parámetros.

```
exp:  NUMBER
    {
      // Create a new number node
      $$ = new lp::NumberNode($1);
    }

  | STRINGS
    {
      $$ = new lp::StringsNode($1);
```

```
  }

| BOOL
  {
    $$ = new lp::BoolNode($1);
  }

| VARIABLE
  {
    // Create a new variable node
    $$ = new lp::VariableNode($1);
  }

| CONSTANT
  {
    // Create a new Constant node

    $$ = new lp::ConstantNode($1);
  }

|   LEFTPARENTHESIS exp RIGHTPARENTHESIS
      {
      // just copy up the expression node
    $$ = $2;
   }

  |   PLUS exp %prec UNARY
  {
    // Create a new unary plus node
      $$ = new lp::UnaryPlusNode($2);
  }

|   MINUS exp %prec UNARY
  {
    // Create a new unary minus node
      $$ = new lp::UnaryMinusNode($2);
  }
```

```
|   exp PLUS exp
  {
    // Create a new plus node
     $$ = new lp::PlusNode($1, $3);
   }

|   exp MINUS exp
      {
    // Create a new minus node
    $$ = new lp::MinusNode($1, $3);
  }

|   exp PRODUCT exp
  {
    // Create a new multiplication node
    $$ = new lp::MultiplicationNode($1, $3);
  }

|   exp DIVISION exp
  {
    // Create a new division node
    $$ = new lp::DivisionNode($1, $3);
   }

| exp INTEGER_DIVISION exp
  {
    $$ = new lp::IntegerDivisionNode($1, $3);
  }

| exp MODULUS exp
  {
    // Create a new modulo node
    $$ = new lp::ModuloNode($1, $3);
     }

| exp POWER_OF exp
    {
    // Create a new power node
```

```
      $$ = new lp::PowerNode($1, $3);
  }

| exp CONCATENATE exp
  {
    $$ = new lp::ConcatenateNode($1, $3);
  }

| exp GREATER_THAN exp
  {
    // Create a new "greater than" node
    $$ = new lp::GreaterThanNode($1,$3);
  }

| exp GREATER_OR_EQUAL exp
  {
    // Create a new "greater or equal" node
    $$ = new lp::GreaterOrEqualNode($1,$3);
  }

| exp LESS_THAN exp
  {
    // Create a new "less than" node
    $$ = new lp::LessThanNode($1,$3);
  }

| exp LESS_OR_EQUAL exp
  {
    // Create a new "less or equal" node
    $$ = new lp::LessOrEqualNode($1,$3);
  }

| exp EQUAL exp
  {
    // Create a new "equal" node
    $$ = new lp::EqualNode($1,$3);
  }
```

```
| exp NOT_EQUAL exp
{
  // Create a new "not equal" node
  $$ = new lp::NotEqualNode($1,$3);
}

| exp AND exp
{
  // Create a new "logic and" node
  $$ = new lp::AndNode($1,$3);
}

| exp OR exp
{
  // Create a new "logic or" node
  $$ = new lp::OrNode($1,$3);
}

| NOT exp
{
  // Create a new "logic negation" node
  $$ = new lp::NotNode($2);
}

| BUILTIN LEFTPARENTHESIS listOfExp RIGHTPARENTHESIS
{
  lp::Builtin *f = (lp::Builtin *) table.getSymbol($1);

  if(f->getNParameters() == (int) $3->size()){
    switch(f->getNParameters()){
      case 0:
        {
          $$ = new lp::BuiltinFunctionNode_0($1);
        }
        break;

      case 1:
        {
```

```
            lp::ExpNode *e = $3->front();

            $$ = new lp::BuiltinFunctionNode_1($1, e);
          }
          break;

        case 2:
          {
            lp::ExpNode *e1 = $3->front();
            $3->pop_front();
            lp::ExpNode *e2 = $3->front();

            $$ = new lp::BuiltinFunctionNode_2($1, e1, e2);
          }
          break;

        default:
          execerror("Syntax error: too many parameters for
    function ", $1);
      }
    }
    else{
      execerror("Syntax error: incompatible number of parameters
    for function", $1);
    }
  }
;
```

Estas son las diferentes reglas de la regla de producción del símbolo no terminal asgn. En la primera de ellas, el símbolo no terminal se sustituye por el símbolo terminal VARIABLE, el terminal ASSIGNMENT y el símbolo no terminal exp. En este caso se crea un nodo de asignación.

En la segunda regla, el símbolo no terminal se sustituye por el símbolo terminal VARIABLE, el terminal ASSIGNMENT y el no terminal asgn, permitiendo así la creación de un nodo de múltiple asignación.

En la tercera regla, el no terminal se sustituye por el terminal VARIABLE,

el terminal SUM_VARIABLE y el no terminal exp, creándose así un nodo de SumVariableStmt.

En la cuarta regla, el no terminal se sustituye por el terminal VARIABLE, el terminal SUBSTRACT_VARIABLE y el no terminal exp, creándose así un nodo de SubstractVariableStmt.

En la quinta regla, el no terminal se sustituye por el terminal VARIABLE, el terminal PRODUCT_VARIABLE y el no terminal exp, creándose así un nodo de ProductVariableStmt.

En la sexta regla, el no terminal se sustituye por el terminal VARIABLE, el terminal DIVIDE_VARIABLE y el no terminal exp, creándose así un nodo de DivideVariableStmt.

En la séptima regla, el no terminal se sustituye por el terminal VARIABLE y el terminal PLUS_PLUS, creándose el nodo PlusPlusStmt. Ocurre lo mismo cambiando el orden de estos terminales. También ocurre igual con MINUS_MINUS, pero en este caso creándose el nodo MinuMinusNode.

Para poder controlar errores, no se ha permitido que a las constantes se le reasigne su valor, como se puede ver en la dos últimas reglas.

```
asgn: VARIABLE ASSIGNMENT exp
  {
    $$ = new lp::AssignmentStmt($1, $3);
  }

  | VARIABLE ASSIGNMENT asgn
  {
    $$ = new lp::AssignmentStmt($1, (lp::AssignmentStmt *) $3);
  }

  | VARIABLE SUM_VARIABLE exp
  {
    $$ = new lp::SumVariableStmt($1, $3);
  }

  | VARIABLE SUBSTRACT_VARIABLE exp
  {
```

```
  $$ = new lp::SubstractVariableStmt($1, $3);
}

| VARIABLE PRODUCT_VARIABLE exp
{
  $$ = new lp::ProductVariableStmt($1, $3);
}

| VARIABLE DIVIDE_VARIABLE exp
{
  $$ = new lp::DivideVariableStmt($1, $3);
}

| VARIABLE PLUS_PLUS
{
  $$ = new lp::PlusPlusStmt($1);
}

| PLUS_PLUS VARIABLE
{
  $$ = new lp::PlusPlusStmt($2);
}

| VARIABLE MINUS_MINUS
{
  $$ = new lp::MinusMinusStmt($1);
}

| MINUS_MINUS VARIABLE
{
  $$ = new lp::MinusMinusStmt($2);
}

| CONSTANT ASSIGNMENT exp
{
  execerror("Semantic error in assignment: it is not allowed to
  modify a constant ", $1);
}
```

```
  | CONSTANT ASSIGNMENT asgn
  {
    execerror("Semantic error in multiple assignment: it is not
    allowed to modify a constant ",$1);
  }
;
```

Estas son las diferentes reglas de producción para el bucle para. En la primera de ellas el no terminal para se sustituye por el terminal FOR, el terminal VARIABLE, el terminal FROM, el no terminal exp, el terminal UNTIL, el no terminal exp, el terminal DO, el no terminal stmtlist y el terminal END_FOR. En este caso se crea el nodo ForStmt llamando al constructor con 4 argumentos, ya que no ha recibido el incremento en cada iteración.

En la segunda regla, lo único que cambia respecto a la primera es que antes del terminal DO, se añade el terminal STEP y el no terminal exp. En este caso, se llama al constructor del nodo ForStmt con 5 argumentos, siendo el último de ellos el que recibe el valor del incremento en cada iteración.

Para controlar errores, en las dos últimas reglas, en vez de poner el terminal VARIABLE, se ha puesto CONSTANT. Como la variable se modifica en cada iteración y las constantes no se pueden cambiar, se lanza un mensaje de error.

```
para: FOR VARIABLE FROM exp UNTIL exp DO stmtlist END_FOR
  {
    $$ = new lp::ForStmt($2, $8, $4, $6);
  }

  | FOR VARIABLE FROM exp UNTIL exp STEP exp DO stmtlist END_FOR
  {
    $$ = new lp::ForStmt($2, $10, $4, $6, $8);
  }

  | FOR CONSTANT FROM exp UNTIL exp DO stmtlist END_FOR
  {
    execerror("Semantic error in for statement: it is not allowed
    to modify a constant ", $2);
```

```
  }

  | FOR CONSTANT FROM exp UNTIL exp STEP exp DO stmtlist END_FOR
  {
    execerror("Semantic error in for statement: it is not allowed
    to modify a constant ", $2);
  }
;
```

La primera regla se refiere a la sentencia de control repetir. El no terminal repetir se sustituye por el terminal REPETITION, el no terminal stmtlist, el terminal UNTIL y el no terminal cond. Así que en este caso se crea el nodo RepeatStmt. La regla de producción del bucle mientras se sustituye por el terminal WHILE, el no terminal cond, el terminal DO, el no terminal stmtlist y el terminal END_WHILE.

Además, hay dos reglas de producción para la sentencia de control if. Se distingue entre que haya ELSE o no. Dependiendo de eso, se llama al constructor con unos atributos u otros.

```
repetir: REPETITION stmtlist UNTIL cond
    {
      $$ = new lp::RepeatStmt($2, $4);
    }
;

mientras: WHILE cond DO stmtlist END_WHILE
    {
      $$ = new lp::WhileStmt($2, $4);
    }
;

si: IF cond THEN stmtlist END_IF
  {
    $$ = new lp::IfStmt($2, $4);
  }

  | IF cond THEN stmtlist ELSE stmtlist END_IF
```

```
  {
    $$ = new lp::IfStmt($2, $4, $6);
  }
;
```

El símbolo no terminal print tiene dos reglas de producción. La diferencia entre estas dos reglas es el primer terminal por el que se sustituye. Por lo tanto la primera regla sera para imprimir expresiones numéricas o booleanas mientras que la segunda regla se encargará de imprimir las expresiones de tipo string.

```
print: WRITE LEFTPARENTHESIS exp RIGHTPARENTHESIS
  {
    $$ = new lp::PrintStmt($3);
  }

  | WRITE_STRING LEFTPARENTHESIS exp RIGHTPARENTHESIS
  {
    $$ = new lp::PrintStringStmt($3);
  }
;
```

En estas reglas de producción para el símbolo no terminal pasa lo mismo que en las reglas para el no terminal print. La primera de ellas leerá variables numéricas mientras que la segunda leerá una variable alfanumérica.

Las dos últimas reglas sirven para controlar errores. En el caso que en vez de poner variables se pongan constantes, aparecerá un mensaje de error.

```
read: READ LEFTPARENTHESIS VARIABLE RIGHTPARENTHESIS
  {
    $$ = new lp::ReadStmt($3);
  }

  | READ_STRING LEFTPARENTHESIS VARIABLE RIGHTPARENTHESIS
  {
    $$ = new lp::ReadStringStmt($3);
  }

  | READ LEFTPARENTHESIS CONSTANT RIGHTPARENTHESIS
```

```
  {
    execerror("Semantic error in for statement: it is not allowed
    to modify a constant ", $3);
  }

  | READ_STRING LEFTPARENTHESIS CONSTANT RIGHTPARENTHESIS
  {
    execerror("Semantic error in for statement: it is not allowed
    to modify a constant ", $3);
  }
;
```

En la regla de producción del símbolo no terminal borrar, este se sustituye por el terminal ERASE, y se crea el nodo EraseStmt.

En la regla del símbolo no terminal lugar, este se sustituye por el terminal PLACE, LEFTPARENTHESIS, NUMBER, COMMA, NUMBER y RIGHTPARENTHESIS. Y en este caso, se crea el nodo PlaceStmt, recibiendo como argumentos el primer y segundo número.

En la regla del no terminal terminar, este se sustituye por el terminal KILL, y se crea el nodo ExitStmt.

Y por último, el no terminal ejecutar se sustituye por el terminal EXECUTE, LEFTPARENTHESIS, STRINGS y RIGHTPARENTHESIS. Y se crea el nodo ExecuteStmt recibiendo como argumento la string.

```
borrar: ERASE
  {
    $$ = new lp::EraseStmt();
  }
;

lugar: PLACE LEFTPARENTHESIS NUMBER COMMA NUMBER RIGHTPARENTHESIS
  {
    $$ = new lp::PlaceStmt($3, $5);
  }
;

terminar: KILL
```

```
  {
    $$ = new lp::ExitStmt();
  }
;

ejecutar: EXECUTE LEFTPARENTHESIS STRINGS RIGHTPARENTHESIS
  {
    $$ = new lp::ExecutionStmt($3);
  }
;
```

En esta regla, el símbolo no terminal cond se sustituye por el terminal LEFTPARENTHESIS, el no terminan exp y el terminal RIGHTPARENTHESIS. Lo único que se copia a $$ es exp.

```
cond: LEFTPARENTHESIS exp RIGHTPARENTHESIS
  {
    $$ = $2;
  }
;
```

Las reglas del símbolo no terminal blocks actúan de manera parecida a las del símbolo no terminal stmtlist. Cuando blocks se sustituye por épsilon, se crea una lista de punteros a objetos de la clase CasesStmt. Cuando se sustituye por el mismo y el terminal VALUE, el no terminal exp, el terminal COLON y el no terminal stmtlist, se crea un objeto de la clase CasesStmt y se inserta al final de la lista. La otra regla que tiene este no terminal, es parecida al anterior, pero en este caso, se le añade los terminales BREAK y SEMICOLON al final. Además se crea un puntero a in objeto del tipo CasesStmt que se inserta en el final de la lista.

En el caso de las reglas de producción de según, en la primera de ellas se sustituiría por el terminal SWITCH, el no terminal cond, el no terminal blocks, los terminales DEFAULT y COLON, el no terminal stmtlist y el terminal END_SWITCH. En la otra regla, se sustituiría por el terminal SWITCH, los no terminales cond y blocks y el terminal END_SWITCH.

```
blocks:
    {
      $$ = new std::list<lp::CasesStmt *>();
    }

    | blocks VALUE exp COLON stmtlist
    {
      $$ = $1;
      $$->push_back(new lp::CasesStmt($3, $5, false));
    }

    | blocks VALUE exp COLON stmtlist BREAK SEMICOLON
    {
      $$ = $1;
      $$->push_back(new lp::CasesStmt($3, $5, true));
    }
;

segun: SWITCH cond blocks DEFAULT COLON stmtlist END_SWITCH
  {
    $$ = new lp::SwitchStmt($2, $3, $6);
  }

  | SWITCH cond blocks END_SWITCH
  {
    $$ = new lp::SwitchStmt($2, $3);
  }
;
```

El símbolo no terminal listOfExp es igual que el no terminal stmtlist, pero en este caso, lo que contiene la lista son punteros a objetos de la clase ExpNode. Cuando se sustituye por épsilon, se crea una nueva lista. Cuando se sustituye por los no terminales exp y restOfListOfExp, el primero se inserta por en la lista que define el segundo no terminal.

En las reglas de producción de restOfListOfExp, cuando este no terminal se sustituye por épsilon, se crea una lista con punteros a objetos de la clase

ExpNode. En la otra regla, cuando este se sustituye por el terminal COMMA, el no terminal exp y el mismo, el no terminal exp se introduce en la lista por delante.

```
listOfExp:
  {
    $$ = new std::list<lp::ExpNode *>();
  }

  | exp restOfListOfExp
  {
    $$ = $2;

    $$->push_front($1);
  }
;

restOfListOfExp:
  {
    $$ = new std::list<lp::ExpNode *>();
  }

  | COMMA exp restOfListOfExp
  {
    $$ = $3;

    $$->push_front($2);
  }
;
```

6. AST

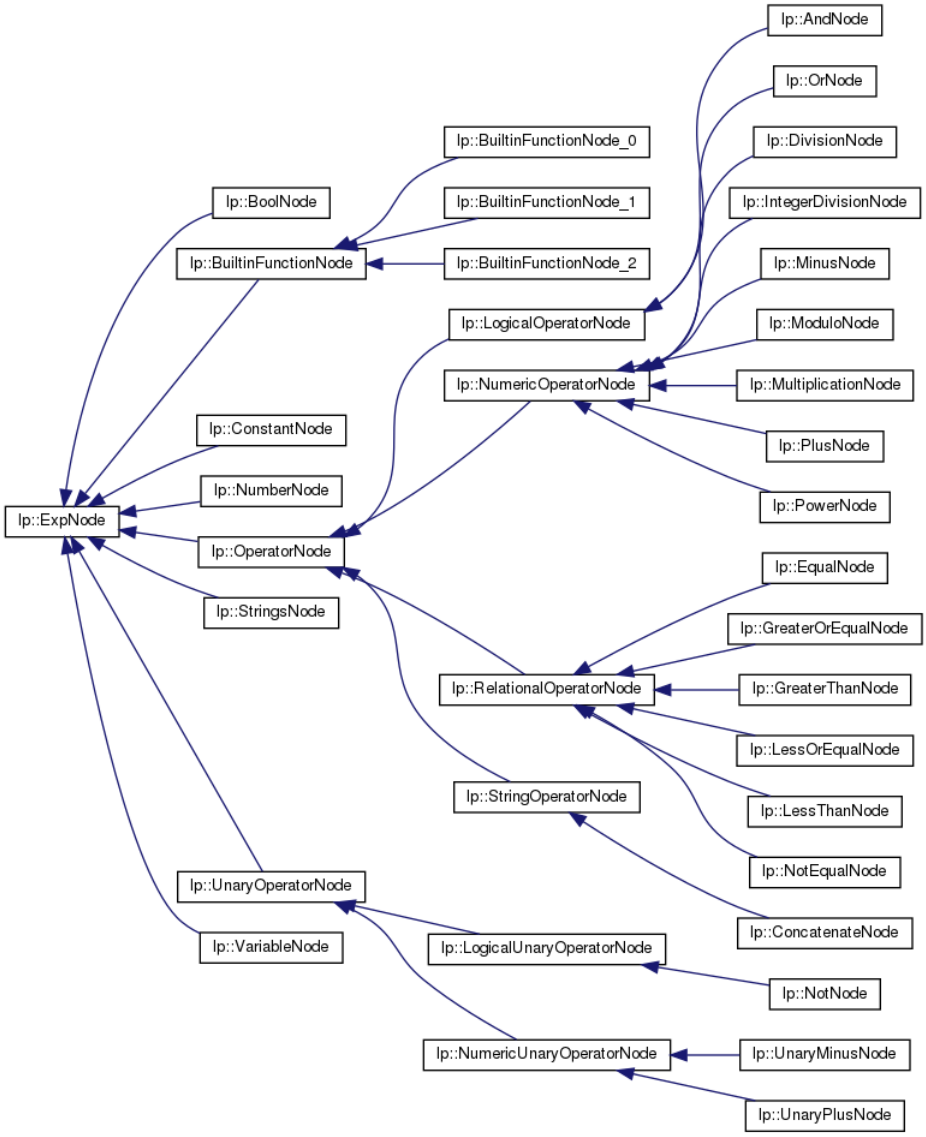

Figura 2: Jerarquía de las clases del AST 1

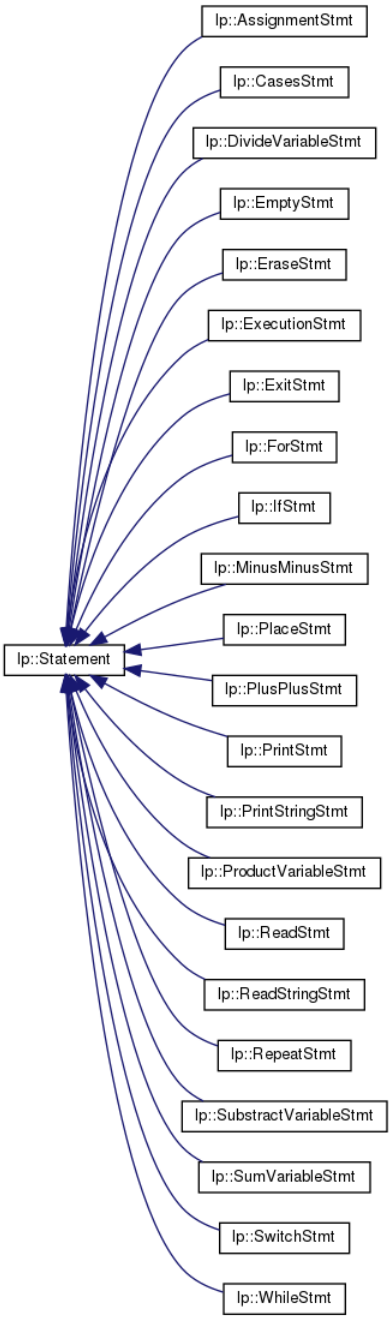

Figura 3: Jerarquía de las clases del AST 2

El conjunto de las clases del árbol de sintaxis abstracto se puede dividir en dos. Aquellas que se basan en la clase abstracta *lp::ExpNode* y aquellas que se basan en *lp::Statement*. Aquellas clases que implementen la clase abstracta *lp::Statement*, como se puede ver en la figura 2, tendrán sus propios observadores y modificadores y redifinirá las funciones virtuales print y evaluate. Mientras que aquellas clases que implementan la clase abstracta *lp::ExpNode*, como se puede ver en la figura 3, tendrán sus propios observadores y modificadores y redifinirá las funciones virtuales print y en su caso evaluateBool, evaluateString ó evaluateNumber.

7. Funciones auxiliares

Las funciones auxiliares que se han implementado son las siguientes:

- **Log**: Calcula el logaritmo neperiano de un número
- **Log10**: Calcula el logaritmo en base 10 de un número
- **Exp**: Calcula el e elevado a un número
- **Sqrt**: Calcula la raíz cuadrada de un número
- **Integer**: Calcula la parte entera de un número decimal.
- ***f***: **¡Esta función es una novedad!** Se ha definido para poder hacer el archivo ejemploOriginal1.e
- **Random**: Calcula un número aleatorio decimal entre 0 y 1.
- **Atan2**: Calcula el arcotangente de un número.
- ***RandomRange***: **¡Esta función es una novedad!** Calcula un número aleatorio decimal e un rango dado por el usuario

8. Modo de obtención del intérprete

Para la obtención del intérprete, se ha definido la siguiente estructura de los ficheros. Se ha dividido en cinco carpetas:

- **ast/**: Esta carpeta contiene las clases que componen el árbol de sintaxis abstracto.
- **error/**: Contiene aquellas funciones que se encargan de manejar la situación del programa en caso de error.
- **includes/**: Contiene el archivo con macros para la visualización en la terminal.
- **parser/**: Esta carpeta incluye el fichero léxico y el fichero con la gramática del lenguaje.
- **table/**: Contiene aquellas clases que van a formar parte de la tabla de símbolos, además de la clase que representa la tabla y el iniciador de esta tabla.

La carpeta **ast/** contiene dos archivos y el makefile. El primero de ellos es *ast.hpp*, el cual contiene las definiciones de las clases del árbol de sintaxis abstracta mientras que el archivo *ast.cpp* contiene la implementación de las diferentes funciones de las diferentes clases de árbol de sintaxis abstracto. La salida de este makefile al invocarlo será un código objeto llamado *ast.o*, el cual ha sido originado al compilar el archivo *ast.cpp* con las banderas de compilación *-c*(opción que genera el código objeto), *-g*(para activar la depuración), *-Wall*(para mostrar todos los avisos), *-ansi*(para cumplir el estándar del lenguaje), *-O2*(para el nivel de optimización) y *-std=c++11*(para el estándar de c++). Además, para obtener esa salida, hacen falta los siguientes headers: *ast.hpp*, *../parser/ipe.tab.h*, *../includes/macros.hpp*, *../error/error.hpp* y *../table/table.hpp*. Cuando se ejecuta el comando *make clean*, se borrará ese código objeto y cualquier backup que haya originado linux (estos archivos terminan en), a ello se debe que se haya puesto la expresión * en el comando rm.

La carpeta **error/** también contiene dos archivos y el makefile. El primero de ellos es *error.hpp*, el cual contiene las cabeceras de las funciones que se encargan de controlar la situación en caso de error. Mientras que el archivo *error.cpp* contiene la implementación de las funciones anteriores. Por último, se encuentra el archivo *makefile*, cuya salida es el archivo con código objeto llamado *error.o*, el cual se ha conseguido compilando el archivo *error.cpp* con las banderas de compilación -*c*(opción que genera el código objeto), -*g*(para activar la depuración), -*Wall*(para mostrar todos los avisos), -*ansi*(para cumplir el estándar del lenguaje), -*O2*(para el nivel de optimización) y -*std=c++11*(para el estándar de c++). Además, para obtener esa salida, hacen falta los siguientes headers: *ast.hpp* y *../includes/macros.hpp*. Cuando se ejecuta el comando *make clean*, se borrará ese código objeto y cualquier backup que haya originado linux (estos archivos terminan en), a ello se debe que se haya puesto la expresión * en el comando rm.

La carpeta **includes/** solo incluye un archivo, el cual es *macros.hpp*. Este archivo contiene las macros para la visualización de pantalla.

La carpeta **parser/** incluye dos archivos y el makefile. El archivo *ipe.l* contiene aquellas expresiones regulares de los elementos clave del código. En el caso de reconocerse una expresión regular u otra, se harán unas acciones u otras. El otro archivo es *ipe.y*, el cual contiene las diferentes reglas de producción de la gramática y en el que se crean los nodos del árbol de sintaxis abstracto. Por último, se encuentra el fichero *makefile*. La salida de este son varios archivos, los cuales son *lex.yy.o*, *lex.yy.c*, *ipe.tab.o*, *ipe.tab.c* y *ipe.tab.h*. Para obtener el archivo *ley.yy.o*, hace falta el archivo *lex.yy.c* y *ipe.tab.h* y se consigue compilando el primero de ellos con las banderas de compilación -*c*(opción que genera el código objeto), -*g*(para activar la depuración), -*Wall*(para mostrar todos los avisos), -*ansi*(para cumplir el estándar del lenguaje), -*O2*(para el nivel de optimización), -*std=c++11*(para el estándar de c++), -*Wno-unused-function*(no mostrar errores relacionados con funciones no utilizados) y -*Wno-sign-compare*. Para obtener el archivo *lex.yy.c*, hacen falta los archivos *ipe.l*, *ipe.tab.h* y las diferentes cabeceras, las cuales son: *../error/error.hpp*, *../in-*

cludes/macros.hpp, *../table/table.hpp*, *../table/numericVariable.hpp*, *../table/init.hpp*, *../table/stringVariable.hpp*, *../table/logicalVariable.hpp*, *../table/numericConstant.hpp*, *../table/builtinParameter0.hpp*, *../table/builtinParameter1.hpp* y *../table/builtinParameter2.hpp*. Para obtenerlo, se llama al comando *flex* y se le pasa el archivo *ipe.l*. Para obtener el archivo *ipe.tab.o*, se necesitan los archivos *ipe.tab.c*, *ipe.tab.h* y las cabeceras anteriormente comentadas. Compilando el archivo *ipe.tab.c* con las banderas de compilación anteriormente mencionadas, se obtiene el código objeto *ipe.tab.o*. Para obtener el archivo *ipe.tab.c* y *ipe.tab.h* hacen falta los archivos *ipe.y* y las cabeceras anteriormente mencionadas. Llamando al comando *bison* con las banderas -*d*(Genera el fichero y.tab.h), -*t*(Para permitir la depuración) y -*g*(para generar el fichero ipe.dot). Y así se obtiene los archivos requeridos. Por último, si se ejecutará el comando *make clean*, se borran los archivos con código objeto, el archivo *ipe.tab.c*, el archivo el archivo *ipe.tab.h*, el archivo *lex.yy.c*, el archivo *ipe.dot*, el archivo *ipe.output* y los distintos backup que hubiese.

La carpeta **table/** esta compuesta por las distintas clase que definen los distintos símbolos que puede haber en la tabla, además de dos ficheros que definen funciones matemáticas. Tambiéne existen dos archivos que se encargan de la inicialización de la tabla de símbolos. La salida principal de este makefile es *table.o*. Para obtenerlo, necesitamos todos los código objeto de todas las clases que ese encuentren en esta carpeta.

En la raíz del proyecto se encuentran los ficheros de Doxygen, *ipe.cpp* y el makefile. El archivo *ipe.cpp* se encarga de iniciar la tabla de símbolos además de decirle al sistema que cuando ocurra un error específico, se manejará con unas de las funciones que se han declarado en *error.hpp*. El makefile de la raíz del proyecto se encarga de compilar el archivo *ipe.cpp* y se encarga de llamar a los distintos makefiles que hay en la distintas carpetas.

9. Modo de ejecución

Existen dos modos de ejecución de este intérprete:

- **A partir de un fichero**: Para ejecutar de esta manera el intérprete hay que pasarle un argumento a este, que representa el nombre del fichero que se quiere ejecutar. Para ello, se tienen que cumplir dos condiciones. La primera de ellas es que la extensión de ese fichero sea .e, de lo contrario no se ejecutará y se imprimirá un mensaje de error por pantalla, terminando la ejecución del intérprete. La segunda condición es la existencia del fichero. En el caso que se le pase por argumento un fichero que no exista, se imprimirá un mensaje de error por pantalla y se terminará la ejecución del intérprete.

- **Interactiva**: **¡Se ha hecho una modificación!** Para ejecutar el intérprete de manera interactiva, solo hay que ejecutarlo sin pasarle ningún archivo como argumento. Una vez ejecutado, se mostrará un "prompt"por pantalla, para que se puedan ejecutar las distintas sentencias. Para poder salir del intérprete, hay que pulsar las teclas Ctrl+D. La modificación que se ha realizado permite que las sentencias de control como el if, for, while, etc, puedan ejecutarse correctamente, ya que anteriormente, el intérprete evaluaba directamente lo que se le pasaba y no esperaba a que se completase la sentencia de control completa.

10. Ejemplos

```
#
   El siguiente algoritmo se encarga de hacer la integración
   de un número.

   Para ello, se utilizará la función x^3 - 1. Para poder
    representar
   esta función, se utilizarán los builtinFuncion, que recibirá un
   argumento, el cual será el valor de x y devolverá el valor de la
     función
   en ese punto.

   Para poder realizar la integración numérica, hay que crear una
    función que
   genere un número aleatorio en un rango.
#

escribir_cadena('Se va a proceder a calcular la integral de x^3 -
    1\n');

escribir_cadena('Introduzca el número de simulaciones: ');
leer(simulaciones);

escribir_cadena('Introduzca los intervalos para el método de los
    trapecios: ');
leer(intervalos);

escribir_cadena('Introduzca el límite inferior de la integral: ');
leer(inferior);

escribir_cadena('Introduzca el límite superior de la integral: ');
leer(superior);

@ Método integración numérica

suma := 0;
```

```
para i desde 1 hasta simulaciones hacer
   x := randomrange(inferior, superior);
   suma +:= f(x);
fin_para;

area := ((superior - inferior)*suma)/simulaciones;


@ Método de los trapecios

step := (superior-inferior)/intervalos;
suma1 := f(inferior) + f(superior);

para j desde 1 hasta intervalos hacer
   suma1 +:= 2 * f(inferior + step*j);
fin_para;

area1 := (suma1 * step) / 2;


escribir_cadena('La integral de la función en esos límites es: ');
escribir(area);
escribir_cadena('\n');

escribir_cadena('La integral de la función utilizando el método de
    los trapecios es: ');
escribir(area1);
escribir_cadena('\n');
```

```
#
   Este programa se encarga de calcular un número combinatorio
#

escribir_cadena('Introduzca m -> ');
leer(m);

escribir_cadena('Introduzca n -> ');
leer(n);

segun(n)
   valor 0:
      resultado := 1;
      romper;

   valor m:
      resultado := 1;
      romper;

   defecto:
      si (m < n) entonces
         escribir_cadena('Es imposible calcular ese número
    combinatorio\n');
         _terminar;
      fin_si;

      numerador := denominador := 1;
      si (m - n < n) entonces
         para i desde n + 1 hasta m hacer
            numerador *:= i;
         fin_para;

         para j desde 1 hasta m - n hacer
            denominador *:= j;
         fin_para;
      si_no
         para i desde m - n + 1 hasta m hacer
            numerador *:= i;
```

```
            fin_para;

            para j desde 1 hasta n hacer
                denominador *:= j;
            fin_para;
        fin_si;

        resultado := numerador / denominador;

fin_segun;

escribir_cadena('El resultado del número combinatorio es: ');
escribir(resultado);
escribir_cadena('\n');
```

11. Conclusiones

El hecho de realizar este trabajo me ha hecho saber como realmente estos intérpretes trabajan por dentro y extrapolar este funcionamiento a lenguajes como python o php. Creo que esta es una buena manera para cometer menos errores cuando estás escribiendo código en el sentido que cuando se está escribiendo una sentencia pensar razonadamente por qué un elemento de ese lenguaje va a continuación o no.

Pienso que los puntos fuertes de este lenguaje son los nuevos elementos y comandos que se han añadido al intérprete como los operadores +:=, -:=, etc. Además se han añadido dos comandos interesantes, uno de ellos para terminar la ejecución de un programa o llamar a otro programa desde uno que se está ejecutando.

Los puntos débiles que encuentro sobre este intérprete es lo poco amigable que es el modo interactivo. Un buen ejemplo de modo interactivo es el de python. Otro punto débil es que no hay estructuras de datos como un vector o una matriz. Por último, una gran mejora hubiera sido la inclusión y definición de funciones.

Referencias

[1] Stack Overflow [sitio web]. [Última Consulta: 27 mayo 2019]. Disponible en: http://www.stackoverflow.com

[2] Moodle Procesadores de Lenguajes [sitio web]. [Última Consulta: 28 mayo 2019]. Disponible en: https://moodle.uco.es/m1819/course/view.php?id=1292