DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP - Graficador de Curvas Mylanga

02/07/2014 Teoria de Lenguajes

Integrante	LU	Correo electrónico
Emilio Almansi		
Santiago Etchegoyen	272/04	santiagoe@fibertel.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1 Introduccion

Se debe programar un compilador que parsee y ejecute código perteneciente al lenguaje "MyLanga". Un programa bien formado en dicho lenguaje es una secuencia de definiciones de funciones, seguida de una última sentencia de ploteo. Dicha sentencia define un rango similar a un *for* de una variable (o de un rango de Smalltalk, ya que define un desde-hasta y un step para cada iteración) y toma dos funciones como parámetro que son las que generan un x,y en cada iteración. Estos puntos son devueltos para graficarse mediante gnuplot.

El lenguaje tiene una sintaxis similar a *C*. En particular los operadores lógicos y aritméticos se pueden considerar idénticos. Al igual que *while* e *if* y sus guardas. Las diferencias más notables con respecto a *C* serían:

- Falta de ; al final de cada instrucción.
- Posibilidad de definir funciones sin llaves para el cuerpo cuando el mismo es de una sola instrucción.
- Cuando el cuerpo del if tiene una sola instrucción se agrega el keyword **then** si se desea evitar llaves.
- Constante **pi**.
- Las variables no se declaran ni se especifica estáticamente su tipo.

Otro problema a tener en cuenta es la detección de errores y la devolución de mensajes declarativos cuando los hubiera.

2 Resolución y aclaraciones

2.1 Generalidades

Para la resolución del problema planteado se decidió utilizar dos de las herramientas propuestas en clase. El analizador léxico **Flex** se utilizó para especificar los tokens del lenguaje MyLanga, generando un *scanner* que transforma el código a procesar en un stream de tokens. Estos tokens pasan a ser los terminales de la gramática que se utiliza para describir con precisión el lenguaje, a partir de la cual el generador de parsers **Bison** construye un parser para MyLanga.

La definición de la gramática que se provee como entrada a **Bison**, asociando a cada producción una regla semántica, la cual genera la lógica necesaria para construir el árbol de sintaxis, o *AST*, que representa el programa procesado.

De esta forma, el intérprete de MyLanga desarrollado utiliza el parser generado de forma automática, y posteriormente ejecuta el código codificado en el *AST*.

Algunas aclaraciones de la implementación:

- Además de los nodos del árbol que representan un grupo de funciones, una instrucción, un if, etc, hay un objeto global que contiene la información contextual del programa, cumpliendo un propósito doble: por un lado, contiene un diccionario con las funciones definidas en el programa, y por el otro, se encarga de generar los scopes durante la ejecución. El mismo se comporta como un stack y se utiliza para saber que variables son visibles tanto a la hora de parsear como ejecutar el código. Cuando se realza un llamado a función, se pushea un estado nuevo al stack dejando vacía la definición de variables (excepto por los parámetros si hay alguno). Una vez que se retorna de la función, se popea el stack para volver al estado inicial del llamador.

- Una vez generado el *AST*, se realiza un proceso de verificación de validez sobre el programa. El mismo consiste en recorrer recursivamente el árbol, evaluando en cada nodo si este es válido, así como también sus hijos. El procedimiento se propaga independientemente de si se encuentra efectivamente algún error, permitiendo detectar múltiples problemas en una única pasada. Posteriormente, bajo la condición de que todos los nodos del árbol verificaran su condición de validez, se procede a ejecutar el comado plot.

- Se agregó la compatibilidad con código que hace recursión explícita. Esto se logró definiendo una función inmediatamente luego de leer su signatura, previo a evaluar la validez de su bloque de código correspondiente. Naturalmente, si se encuentra algún error en el cuerpo de la función, se elimina la misma de la tabla de contexto, generando posiblemente una serie de errores en el resto del programa donde la misma sea utilizada.

2.2 Requisitos y modos de uso

- La implementación requiere tener instaladas las herramientas *Flex* y *Bison*.

- La implementación del AST fue escrita en C++ standard 11. En el caso de usar g++, el mismo debe ser versión 4.7 o superior, para soportar la directiva -std=c++11. No es compatible con el std c++0x.

- El script mylanga_plot.sh grafica un archivo mylanga mediante la siguiente sintaxis:

  ./mylanga_plot.sh programa.my salida.png

  Ejemplo: ./mylanga_plot.sh circulo.my circulo.png

- También se proveen los scripts correr-tests-correctos.sh y correr-tests-errores.sh para automatizar la corrida de los tests provistos por la cátedra.

2.3 Gramática

Definimos la siguiente gramática G = < { Program, Seq_fun_def, Fun_def, Lst_id, Lst_id2, Block, Seq_stmt, Stmt, Pred, Expr, Lst_expr, Lst_expr2, Plot_cmd }, { for, plot, if, then, else, while, return, function, pi, id, int, float, +, −, ∗, /, ∧, , ,, .., =, ||, &&, !, <, <=, ==, >=, >, (,), {, } }, P, Program > donde P son las siguientes producciones:

Program → Seq_fun_def Plot_cmd
Una programa válido es una secuencia de funciones y un comando plot.

Seq_fun_def → Fun_def | Seq_fun_def Fun_def
Una secuencia de funciones es una función o una secuencia seguida de una función.

Fun_def → function id (Lst_id) Block
Una función bien definida es un ID de función seguido de los parámetros (lista de ids) rodeados por paréntesis y por último un bloque de instrucciones.

Lst_id → λ | Lst_id2
Una lista de parámetros es o bien una lista vacía o una con parámetros.

Lst_id2 → id | Lst_id2 , id
Una lista con parámetros es o un ID o una lista seguida de una coma y otro ID.

Block → Stmt | { Seq_stmt }
Un bloque de instrucciones es o bien una única instrucción o una secuencia de instrucciones entre llaves.

Seq_stmt → Stmt | Seq_stmt Stmt
Una secuencia de instrucciones es o bien una instrucción o una secuencia seguida de una instrucción.

Stmt → id = Expr
| if Pred then Block
| if Pred then Block else Block
| while Pred block
| return Expr
Las instrucciones son o bien una asignación, un if (con y sin else), un while o un return.

Pred → Expr < Expr
| Expr <= Expr
| Expr == Expr
| Expr >= Expr
| Expr > Expr
| Pred || Pred
| Pred && Pred
| ! Pred
| (Pred)
Los predicados se componen mediante operadores binarios o negaciones. También pueden estar rodeados por paréntesis.

Expr → int
| float
| pi
| id
| Expr + Expr
| Expr - Expr
| Expr * Expr
| Expr / Expr
| Expr ∧ Expr
| - Expr
| id (Lst_expr)
| (Expr)
Las expresiones son o bien literales de enteros y flotantes, o un id de variable, o una operación aritmética entre expresiones. También pueden ser llamados a funciones, la negativización, o la constante pi. Por último, también pueden estar rodeadas por paréntesis.

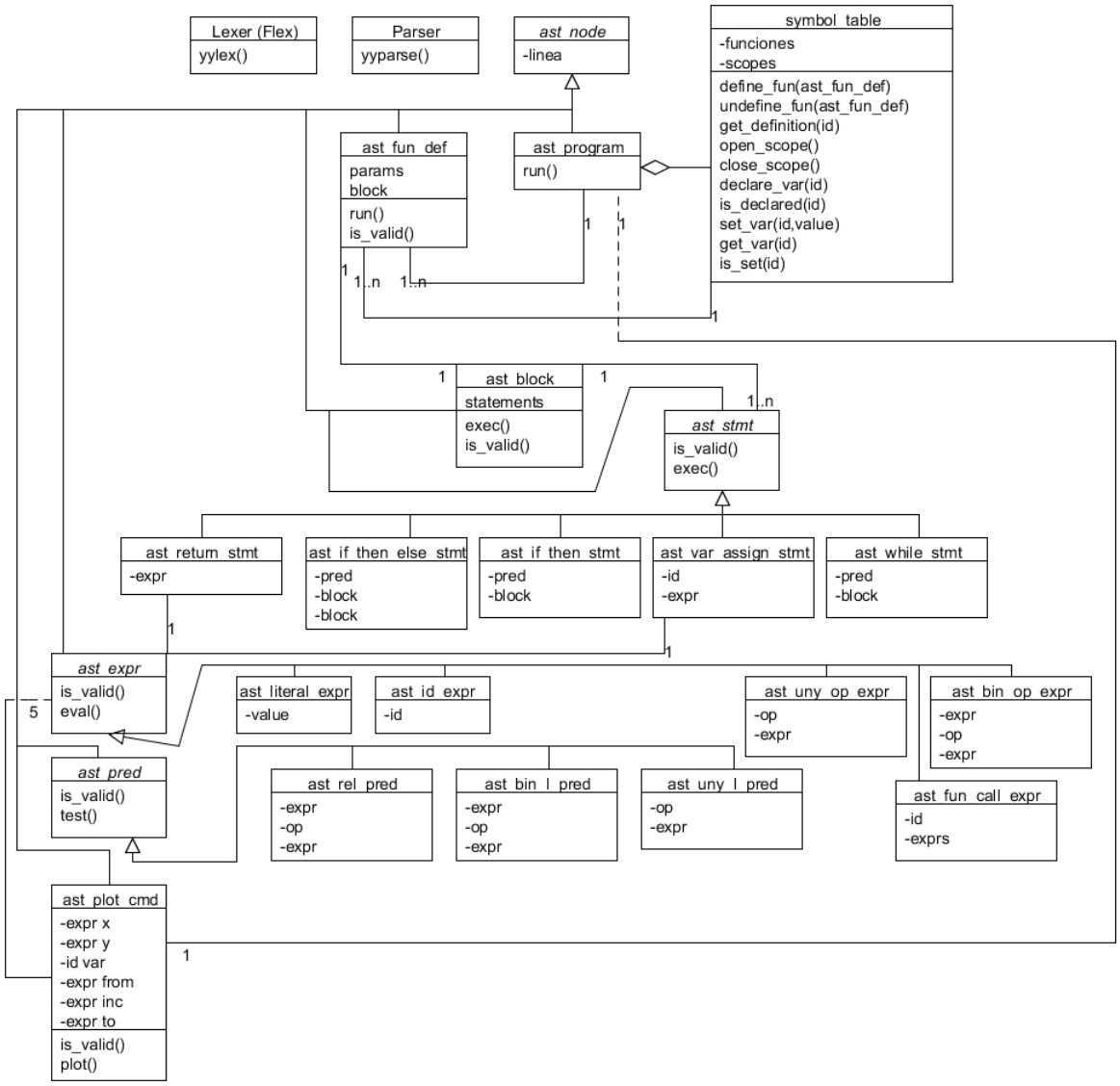
Lst_expr → λ | Lst_expr2
Una lista de expresiones es o bien una lista vacía o una con expresiones.

Lst_expr2 → Expr | Lst_expr2 , Expr
Una lista con expresiones es o una expresión o una lista seguida de una coma y otra expresión.

Plot_cmd → plot (Expr , Expr) for id = Expr .. Expr .. Expr
Por último el comando plot recibe dos expresiones a llamar y un for con 3 expresiones con un valor, un tope y un incremento para una variable id.

2.4 Arquitectura

2.4.1 Clases



En este diagrama de clases podemos observar la descripción de las clases más importantes de la resolución. En primer lugar tenemos al parser y el lexer, representando a Bison y a Flex que fueron elegidos a la hora de la resolución. Luego tenemos la tabla de símbolos utilizada para almacenar tanto el stack de variables en cada contexto de ejecución como las funciones definidas. Su comportamiento se verá más en detalle en las secuencias siguientes, a la hora de explicar tanto el proceso de validación como de ejecución y la importancia de este objeto.

Luego tenemos los nodos AST, representados por una clase abstracta de la cual todos derivan. En particular la gran mayoría saben responder métodos de validación y de ejecución o evaluación. Nuevamente, esto se verá mejor explicado en las próximas secuencias.

A priori todo árbol comienza con un nodo de tipo programa que tiene un grupo de funciones definidas y un statement de ploteo. A su vez cada función tiene un bloque, y un bloque tiene statements. Los statements representan las estructuras de control de flujo de programa y las asignaciones. Por lo tanto, estas según la clase pueden tener a su vez bloques con statements,

predicados o expresiones. En el esquema no se pusieron todas las relaciones entre statements predicados y expresiones ya que entorpecía la lectura. Solo se conservaron en el esquema las flechas de jerarquías y algunas relaciones esenciales. Para analizar el resto hay que mirar en detalle los atributos de la clase.

Las expresiones son literales, ids de variables u operadores sobre otras expresiones. Todas responden a la evaluación y chequeo de validez, al igual que los statements. Ya se puede suponer entonces la naturaleza recursiva de la implementación.

Por último los predicados al igual que las expresiones combinan operadores con expresiones, pero esta vez para evaluar condiciones binarias.

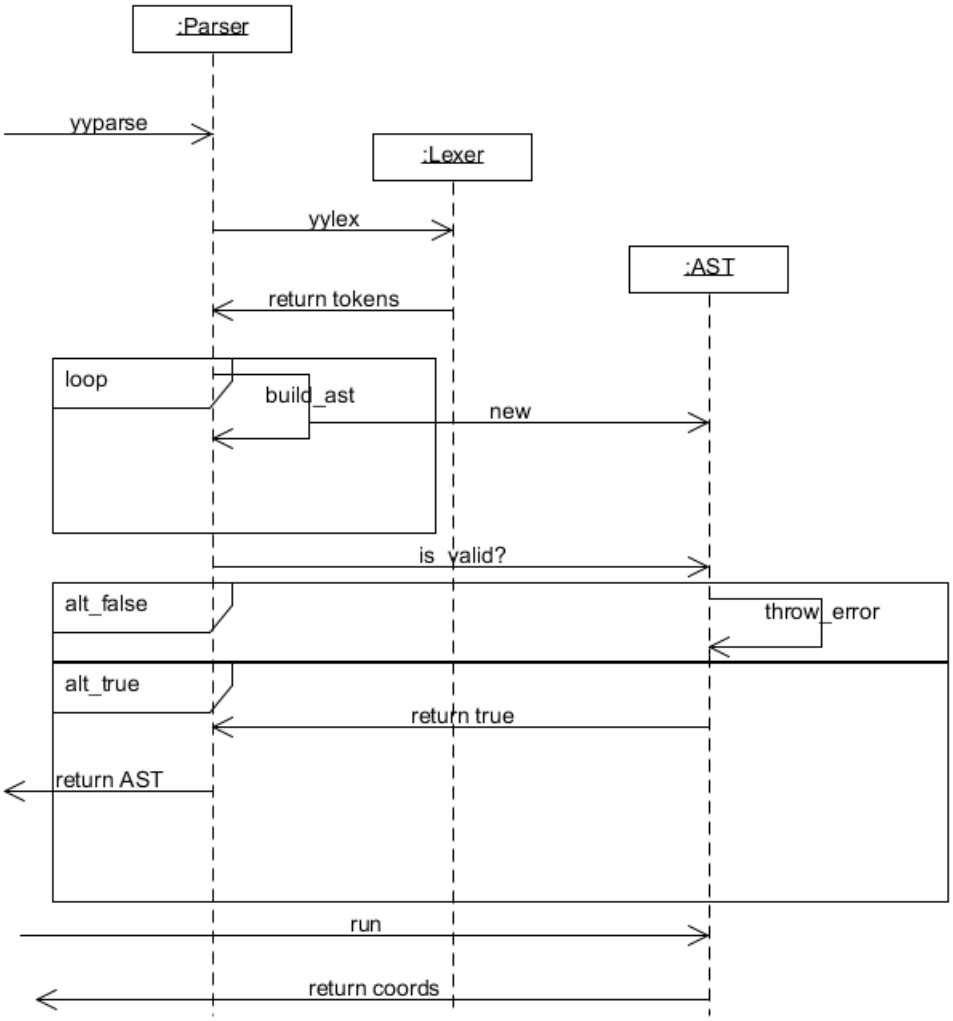
Por último la clase que representa al comando plot contiene las expresiones que generan el x e y, el id de la variable que se mueve en el rango a plotear, y las expresiones que calculan tanto el inicio como el incremento y el tope. Cabe recordar que en nuestra implementación el poder usar expresiones en lugar de llamados a funciones es una particularidad elegida para extender el lenguaje.

2.4.2 Segumiento de prueba

En las próximas secuencias intentaremos explicar como se relacionan estas clases mediante el seguimiento de este código simple de prueba:

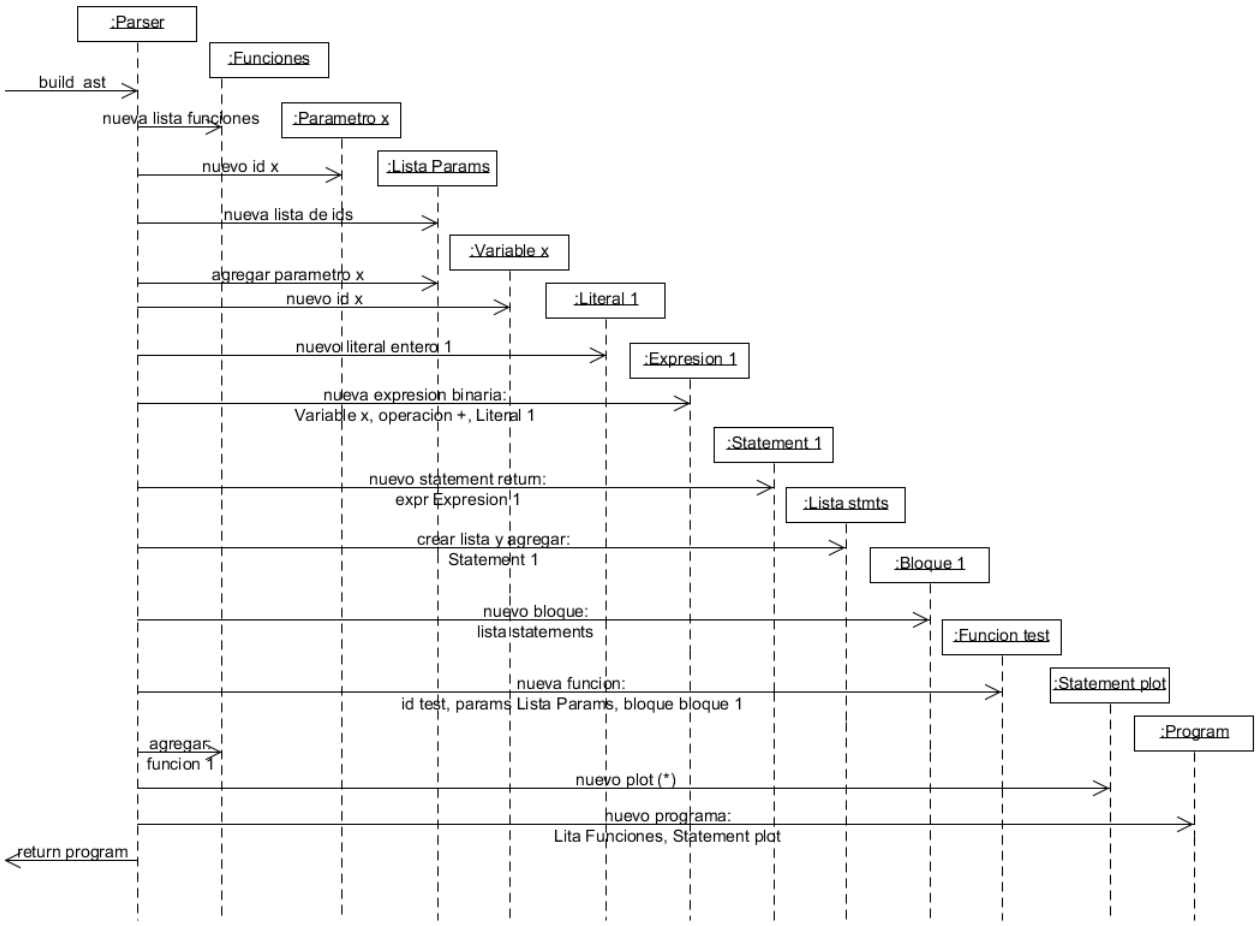
```
function test(x)
        return x+1

plot(test(x),test(x)) for x=1..1..6
```

2.4.3 Secuencia global



La idea de esta secuencia es mostrar el funcionamiento a más alto nivel del compilador MyLanga. El parser le pide los tokens al lexer, y mediante sus reglas semánticas genera un AST basado en el código recibido. Luego el mismo pasa por una secuencia de validación donde más adelante veremos que se controlan una serie de condiciones, y finalmente se ejecuta para devolver las coordenadas a dibujar.

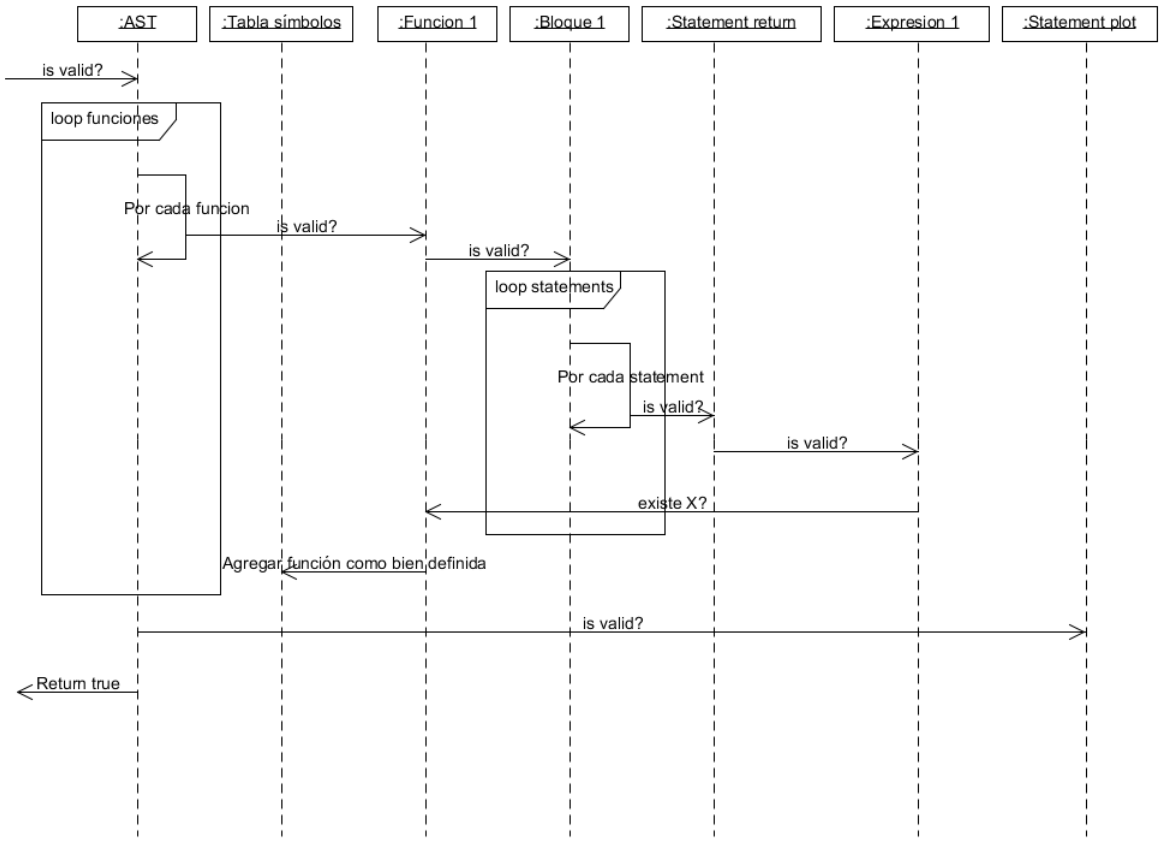
2.4.4 Secuencia de generación del AST



En esta secuencia vemos como se va armando el AST para el código de ejemplo. Para esto el parser genera instancias nuevas para X y el literal 1, y los junta en una expresion X+1. Esa expresión a su vez se incluye en un statement de retorno y pasa a formar un bloque de statements. Dicho bloque se agrega a una declaración de función junto a una lista de ids que son los parámetros, y dicha declaración se agrega a la lista de funciones declaradas. Esa lista más un comando plot pasan a formar un programa, y el árbol se completa.

Para conservar la legibilidad del esquema no se puso como se forma el comando plot, ya que implica instanciar 5 expresiones y un id. Pero el comportamiento es idéntico al descripto anteriormente y se terminan agrupando en el nodo que lo representa.

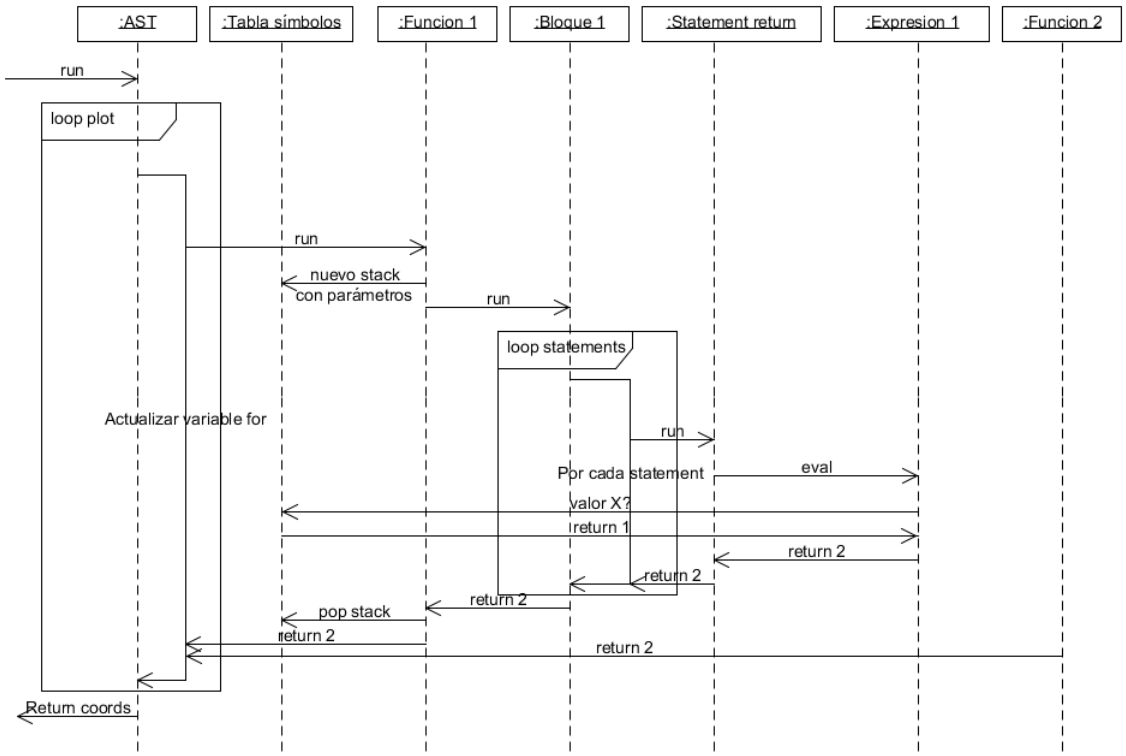
2.4.5 Secuencia de validación del AST



En esta secuencia vemos la etapa de validación, que básicamente propaga recursivamente por todo el árbol un llamado de chequeo de validez de cada nodo. Cada tipo de nodo tiene sus reglas para verificar dicha condición y recibe paramétricamente la tabla única de símbolos para realizar la tarea. En un primer lugar el programa le pide a cada una de sus funciones que se chequeen. Estas a su vez le piden a sus bloques que lo hagan. El bloque itera cada statement para pedirle a su vez la misma operación. Si todo está bien se pasa a chequear el comando plot que es la última instancia a considerar para verificar errores.

Cada expresión tiene un set de reglas para verificar validez. Por ejemplo un llamado a una función deberá verificar que la función esté definida. Si no está definida, a su vez puede ser un llamado recursivo y debiera ser válido también. (Esto no se ve en este ejemplo). Si es una expresión sobre un Id de variable, esta debe estar definida en el scope actual. Si una función pasa los chequeos, la misma debiera ser tenida en cuenta para futuros llamados. Todas estas cuestiones solo son posibles mediante la propagación recursiva del objeto de tabla de símbolos que almacena todas estas informaciones.

2.4.6 Secuencia de ejecución del AST



La idea en la ejecución es similar a la de la evaluación. El código se ejecuta recursivamente hasta llegar a evaluaciones, y esas evaluaciones son integradas mediante los statements que las invocan. A modo general el programa va a iterar el rango definido en el comando plot, y evaluar las expresiones (llamados a funciones) que generan X e Y.

Una vez que se llama a la función, entramos en un scope nuevo que solo contiene como variables a los parámetros. A su vez cada función va a llamar a la corrida de su bloque, que llamará a la corrida de cada uno de sus statements, que a su vez evaluarán expresiones, que pueden llamar a otras funciones, etc.

En particular algo que no se ve en este esquema es que el statement de asignación escribirá en el stack reemplazando (o definiendo) el valor de una variable. Si podemos ver claramente que para evaluar X+1 se busca el valor de X en el stack.

2.5 Flex

```
%{

#include <string>
#include <iostream>
using namespace std;

#include "mylanga_ast.h"
#include "mylanga_sem_types.h"
#include "mylanga_error.h"
#define YYSTYPE mylanga_sem_types

#include "parser.hpp"

#define ECHO // no borrar, sino tokens.cpp mete \n's espurios en la salida
extern "C" int yywrap() { }
int line_num = 1;

%}

%x IN_SINGL_COMMENT
%x IN_MULTI_COMMENT

%%

<INITIAL>{
"//"                    BEGIN(IN_SINGL_COMMENT);
"/*"                    BEGIN(IN_MULTI_COMMENT);
[ \t]                   ;
\n                      ++line_num;
"for"                   yylval._int = KW_FOR; return KW_FOR;
"plot"                  yylval._int = KW_PLOT; return KW_PLOT;
"if"                    yylval._int = KW_IF; return KW_IF;
"then"                  yylval._int = KW_THEN; return KW_THEN;
"else"                  yylval._int = KW_ELSE; return KW_ELSE;
"while"                 yylval._int = KW_WHILE; return KW_WHILE;
"return"                yylval._int = KW_RETURN; return KW_RETURN;
"function"              yylval._int = KW_FUNCTION; return KW_FUNCTION;
"pi"                    yylval._int = KW_PI; return KW_PI;
[a-zA-Z][a-zA-Z0-9_]*   yylval._id = mp<id>(yytext, yyleng); return ID;
[0-9]+                  yylval._str = mp<string>(yytext, yyleng); return INT_LITERAL;
[0-9]+\.[0-9]+          yylval._str = mp<string>(yytext, yyleng); return FP_LITERAL;
"+"                     yylval._int = OP_PLUS; return OP_PLUS;
"-"                     yylval._int = OP_MINUS; return OP_MINUS;
"*"                     yylval._int = OP_MULT; return OP_MULT;
"/"                     yylval._int = OP_DIV; return OP_DIV;
"^"                     yylval._int = OP_EXP; return OP_EXP;
";"                     yylval._int = SEMICOLON; return SEMICOLON;
","                     yylval._int = COMMA; return COMMA;
".."                    yylval._int = ELLIPSIS; return ELLIPSIS;
"="                     yylval._int = EQUAL; return EQUAL;
"||"                    yylval._int = L_OR; return L_OR;
"&&"                    yylval._int = L_AND; return L_AND;
"!"                     yylval._int = L_NOT; return L_NOT;
"<"                     yylval._int = REL_LT; return REL_LT;
"<="                    yylval._int = REL_LEQ; return REL_LEQ;
"=="                    yylval._int = REL_EQ; return REL_EQ;
">="                    yylval._int = REL_GEQ; return REL_GEQ;
">"                     yylval._int = REL_GT; return REL_GT;
"("                     yylval._int = LPAREN; return LPAREN;
")"                     yylval._int = RPAREN; return RPAREN;
"{"                     yylval._int = LBRACE; return LBRACE;
"}"                     yylval._int = RBRACE; return RBRACE;
.                       cerr << MYLANGA_LEXER_ERROR(line_num) << "␣|␣" << "Token␣no␣
    reconocido␣\"" << string(yytext, yyleng) << "\"." << endl; yyterminate();
}

<IN_SINGL_COMMENT>{
\n                      BEGIN(INITIAL);
.                       ;
```

```
}

<IN_MULTI_COMMENT>{
"*/"                        BEGIN(INITIAL);
.                           ;
}

%%
```

Definimos todos los tokens posibles según el lenguaje MyLanga. Cabe destacar que para hacer comentarios multilínea se hace uso de un cambio de contexto en Flex que solo puede retornar al contexto original una vez que lee el token de fin de comentario.

2.6 Bison

```
%{

#include <list>
#include <iostream>
using namespace std;

#include "mylanga_fp_t.h"
#include "mylanga_ast.h"
#include "mylanga_sem_types.h"
#include "mylanga_error.h"
#define YYSTYPE mylanga_sem_types

extern int line_num; int temp;
extern ptr<ast_program> pg;

extern int yylex();
void yyerror(const char *s) { cerr << MYLANGA_SYNTAX_ERROR(line_num) << endl; }

%}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

/* terminales de la gramática */

%token <_id> ID
%token <_str> INT_LITERAL
%token <_str> FP_LITERAL
%token <_int> KW_FOR KW_PLOT KW_IF KW_THEN KW_ELSE KW_WHILE KW_RETURN KW_FUNCTION KW_PI
%token <_int> EQUAL COMMA SEMICOLON ELLIPSIS
%token <_int> L_OR L_AND L_NOT
%token <_int> REL_EQ REL_LT REL_LEQ REL_GEQ REL_GT
%token <_int> LPAREN RPAREN LBRACE RBRACE
%token <_int> OP_PLUS OP_MINUS OP_MULT OP_DIV OP_EXP

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

/* no terminales de la gramática */

%type <_pg> program
%type <_fd> fun_def
%type <_pc> plot_cmd
%type <_bl> block
%type <_st> stmt
%type <_ex> expr
%type <_pr> pred
%type <_fds> seq_fun_def
%type <_sts> seq_stmt
%type <_ids> lst_id
%type <_ids> _lst_id
%type <_exs> lst_expr
%type <_exs> _lst_expr

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

/* precedencia de operadores */

%right KW_THEN KW_ELSE
%left L_OR
%left L_AND
%left L_NOT
%left OP_PLUS OP_MINUS
%left OP_MULT OP_DIV
%left UMINUS
%left OP_EXP

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

/* símbolo distinguido de la gramática */
```

```
%start program

%%

program
  : seq_fun_def plot_cmd          { pg = $$ = mp<ast_program>($1, $2, line_num); }

  // error handling
  | seq_fun_def                   { pg = $$ = mp<ast_program>($1, mp<ast_syntax_error>("Falta␣
      definir␣la␣instrucción␣de␣plot.", line_num), line_num); }
  | plot_cmd                      { pg = $$ = mp<ast_program>(mp<list<ptr<ast_fun_def>>>(1, mp<
      ast_syntax_error>("No␣se␣definió␣ninguna␣función.", line_num)), $1, line_num); }
  | /* */                         { pg = $$ = mp<ast_program>(mp<list<ptr<ast_fun_def>>>(1, mp<
      ast_syntax_error>("No␣se␣definió␣ninguna␣función.", line_num)), mp<ast_syntax_error>("
      Falta␣definir␣la␣instrucción␣de␣plot.", line_num), line_num); }
  ;

fun_def
  : KW_FUNCTION ID LPAREN lst_id RPAREN block
                                  { $$ = mp<ast_fun_def>($2, $4, $6, line_num); }

  // error handling
  | KW_FUNCTION ID LPAREN lst_id RPAREN /* */
                                  { $$ = mp<ast_syntax_error>("La␣definición␣de␣la␣función␣\'"
                                      + *$2 + "\'␣está␣incompleta.", line_num); }
  ;

plot_cmd
  : KW_PLOT LPAREN expr COMMA expr RPAREN
    KW_FOR ID EQUAL expr ELLIPSIS expr ELLIPSIS expr
                                  { $$ = mp<ast_plot_cmd>($3, $5, $8, $10, $12, $14, line_num);
                                       }

  // error handling
  | KW_PLOT expr
    KW_FOR ID EQUAL expr ELLIPSIS expr ELLIPSIS expr
                                  { $$ = mp<ast_syntax_error>("En␣la␣instrucción␣plot,␣se␣
                                      espera␣un␣par␣(fx(...),fy(...))␣luego␣de␣la␣palabra␣\"
                                      plot\".", line_num); }
  | KW_PLOT LPAREN expr COMMA expr RPAREN /* */
                                  { $$ = mp<ast_syntax_error>("En␣la␣instrucción␣plot,␣falta␣
                                      definir␣un␣rango␣de␣evaluación.", line_num); }
  ;

block
  : stmt                          { $$ = mp<ast_block>(mp<list<ptr<ast_stmt>>>(1, $1), line_num
      ); }
  | LBRACE seq_stmt RBRACE        { $$ = mp<ast_block>($2, line_num); }

  // error handling
  | LBRACE RBRACE                 { $$ = mp<ast_syntax_error>("Un␣bloque␣de␣instrucciones␣no␣
      puede␣ser␣vacío.", line_num); }
  ;

stmt
  : ID EQUAL expr                           { $$ = mp<ast_var_assign_stmt>($1, $3, line_num);
       }
  | KW_IF pred KW_THEN block                { $$ = mp<ast_if_then_stmt>($2, $4, line_num); }
  | KW_IF pred KW_THEN block KW_ELSE block  { $$ = mp<ast_if_then_else_stmt>($2, $4, $6,
      line_num); }
  | KW_WHILE pred block                     { $$ = mp<ast_while_stmt>($2, $3, line_num); }
  | KW_RETURN expr                          { $$ = mp<ast_return_stmt>($2, line_num); }

  // error handling
  | error EQUAL expr                                        { $$ = mp<ast_syntax_error>("
      Instrucción␣de␣asignación␣inválida.", line_num); }
  | ID EQUAL error                                          { $$ = mp<ast_syntax_error>("
      La␣expresión␣de␣lado␣derecho␣de␣la␣asignación␣es␣inválida.", line_num); }
  | KW_IF expr { temp = line_num; } KW_THEN block KW_ELSE block   { $$ = mp<ast_syntax_error>("
      La␣guarda␣de␣un␣if␣debe␣ser␣un␣predicado␣con␣valor␣booleano.", temp); }
  | KW_WHILE expr { temp = line_num; } block                { $$ = mp<ast_syntax_error>("
```

```
      La␣guarda␣de␣un␣while␣debe␣ser␣un␣predicado␣con␣valor␣booleano.", temp); }
  | KW_IF pred { temp = line_num; } block                          { $$ = mp<ast_syntax_error>("
      Falta␣la␣palabra␣clave␣'then'␣en␣la␣instrucción␣if.", temp); }
  | KW_IF error                                                    { $$ = mp<ast_syntax_error>("
      La␣guarda␣del␣if␣es␣inválida.", line_num); }
  | KW_WHILE error                                                 { $$ = mp<ast_syntax_error>("
      La␣guarda␣del␣while␣es␣inválida.", line_num); }
  | KW_RETURN error                                                { $$ = mp<ast_syntax_error>("
      La␣expresión␣de␣retorno␣es␣inválida.", line_num); }
  ;

expr
  : INT_LITERAL                  { $$ = mp<ast_literal_expr>(stofp_t(*$1), line_num); }
  | FP_LITERAL                   { $$ = mp<ast_literal_expr>(stofp_t(*$1), line_num); }
  | KW_PI                        { $$ = mp<ast_literal_expr>(FP_T_PI, line_num); }
  | ID                           { $$ = mp<ast_id_expr>($1, line_num); }
  | expr OP_PLUS expr            { $$ = mp<ast_bin_op_expr>($1, $2, $3, line_num); }
  | expr OP_MINUS expr           { $$ = mp<ast_bin_op_expr>($1, $2, $3, line_num); }
  | expr OP_MULT expr            { $$ = mp<ast_bin_op_expr>($1, $2, $3, line_num); }
  | expr OP_DIV expr             { $$ = mp<ast_bin_op_expr>($1, $2, $3, line_num); }
  | OP_MINUS expr %prec UMINUS   { $$ = mp<ast_uny_op_expr>($1, $2, line_num); }
  | expr OP_EXP expr             { $$ = mp<ast_bin_op_expr>($1, $2, $3, line_num); }
  | ID LPAREN lst_expr RPAREN    { $$ = mp<ast_fun_call_expr>($1, $3, line_num); }
  | LPAREN expr RPAREN           { $$ = $2; }

  // error handling
  | expr OP_PLUS error            { $$ = mp<ast_syntax_error>("Expresión␣inválida.", line_num)
      ; }
  | expr OP_MINUS error           { $$ = mp<ast_syntax_error>("Expresión␣inválida.", line_num)
      ; }
  | expr OP_MULT error            { $$ = mp<ast_syntax_error>("Expresión␣inválida.", line_num)
      ; }
  | expr OP_DIV error             { $$ = mp<ast_syntax_error>("Expresión␣inválida.", line_num)
      ; }
  | expr OP_EXP error             { $$ = mp<ast_syntax_error>("Expresión␣inválida.", line_num)
      ; }
  | LPAREN error RPAREN           { $$ = mp<ast_syntax_error>("Expresión␣inválida.", line_num)
      ; }
  ;

pred
  : expr REL_LT expr             { $$ = mp<ast_rel_pred>($1, $2, $3, line_num); }
  | expr REL_LEQ expr            { $$ = mp<ast_rel_pred>($1, $2, $3, line_num); }
  | expr REL_EQ expr             { $$ = mp<ast_rel_pred>($1, $2, $3, line_num); }
  | expr REL_GEQ expr            { $$ = mp<ast_rel_pred>($1, $2, $3, line_num); }
  | expr REL_GT expr             { $$ = mp<ast_rel_pred>($1, $2, $3, line_num); }
  | pred L_OR pred               { $$ = mp<ast_bin_l_pred>($1, $2, $3, line_num); }
  | pred L_AND pred              { $$ = mp<ast_bin_l_pred>($1, $2, $3, line_num); }
  | L_NOT pred                   { $$ = mp<ast_uny_l_pred>($1, $2, line_num); }
  | LPAREN pred RPAREN           { $$ = $2; }

  // error handling
  | expr REL_LT error             { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | expr REL_LEQ error            { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | expr REL_EQ error             { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | expr REL_GEQ error            { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | expr REL_GT error             { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | pred L_OR error               { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | pred L_AND error              { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  | L_NOT error                   { $$ = mp<ast_syntax_error>("Predicado␣inválido", line_num);
      }
  ;

seq_fun_def
```

```
  : fun_def                         { $$ = mp<list<ptr<ast_fun_def>>>(1, $1); }
  | seq_fun_def fun_def             { $$ = $1; $$->push_back($2); }
  ;

seq_stmt
  : stmt                            { $$ = mp<list<ptr<ast_stmt>>>(1, $1); }
  | seq_stmt stmt                   { $$ = $1; $$->push_back($2); }
  ;

lst_id
  : /* */                           { $$ = mp<list<ptr<id>>>(); }
  | _lst_id                         { $$ = $1; }
  ;

_lst_id
  : ID                              { $$ = mp<list<ptr<id>>>(1, $1); }
  | _lst_id COMMA ID                { $$ = $1; $$->push_back($3); }
  ;

lst_expr
  : /* */                           { $$ = mp<list<ptr<ast_expr>>>(); }
  | _lst_expr                       { $$ = $1; }
  ;

_lst_expr
  : expr                            { $$ = mp<list<ptr<ast_expr>>>(1, $1); }
  | _lst_expr COMMA expr            { $$ = $1; $$->push_back($3); }
  ;

%%
```

Las reglas de bison son muy similares a las de la gramática definida anteriormente. La principal diferencia es que se agregan reglas de error para que matcheen explícitamente algunos casos de error específicos y de esa manera poder devolver mensajes más declarativos. A su vez se ve como se arma el árbol *AST* que luego será utilizado para la ejecución del código. Es decir, al programa se le agrega una instancia de su bloque de funciones y su llamado a plot. Al bloque de funciones se le va empujando cada función definida. Dentro de cada función se agrega cada instrucción, etc.

2.7 Implementación del AST

2.7.1 Definición de clases y estructuras

Primero definimos todas las clases que se usaron para implementar el AST. Se puede ver que son las mismas que se llaman desde las reglas de Bison.

```
#ifndef MYLANGA_AST_H
#define MYLANGA_AST_H

#include "mylanga_fp_t.h"
#include "mylanga_ast_meta.h"

#include <string>
#include <list>
#include <stack>
#include <map>
using namespace std;
typedef string id;

#include <memory>
template <typename T>
using ptr = shared_ptr<T>;
#define mp make_shared

struct ast_expr;
struct ast_literal_expr;
struct ast_id_expr;
struct ast_bin_op_expr;
struct ast_uny_op_expr;
struct ast_pred;
struct ast_rel_pred;
struct ast_bin_l_pred;
struct ast_uny_l_pred;
struct ast_fun_call_expr;
struct ast_stmt;
struct ast_block;
struct ast_var_assign_stmt;
struct ast_if_then_stmt;
struct ast_if_then_else_stmt;
struct ast_while_stmt;
struct ast_return_stmt;
struct ast_plot_cmd;
struct ast_fun_def;
struct ast_program;
struct ast_syntax_error;

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

#define ast_node_fields                 \
  int, _ln

#define ast_program_fields              \
  ptr<list<ptr<ast_fun_def>>>, _fds,    \
  ptr<ast_plot_cmd>, _pc

#define ast_fun_def_fields              \
  ptr<id>, _id,                         \
  ptr<list<ptr<id>>>, _ids,             \
  ptr<ast_block>, _bl

#define ast_plot_cmd_fields             \
  ptr<ast_expr>, _ex_x,                 \
  ptr<ast_expr>, _ex_y,                 \
  ptr<id>, _id,                         \
  ptr<ast_expr>, _ex1,                  \
  ptr<ast_expr>, _ex2,                  \
  ptr<ast_expr>, _ex3

#define ast_block_fields                \
```

```
  ptr<list<ptr<ast_stmt>>>, _sts

#define ast_var_assign_stmt_fields       \
  ptr<id>, _id,                          \
  ptr<ast_expr>, _ex

#define ast_if_then_stmt_fields          \
  ptr<ast_pred>, _pr,                    \
  ptr<ast_block>, _bl

#define ast_if_then_else_stmt_fields     \
  ptr<ast_pred>, _pr,                    \
  ptr<ast_block>, _bl1,                  \
  ptr<ast_block>, _bl2

#define ast_while_stmt_fields            \
  ptr<ast_pred>, _pr,                    \
  ptr<ast_block>, _bl

#define ast_return_stmt_fields           \
  ptr<ast_expr>, _ex

#define ast_rel_pred_fields              \
  ptr<ast_expr>, _ex1,                   \
  int, _op,                              \
  ptr<ast_expr>, _ex2

#define ast_bin_l_pred_fields            \
  ptr<ast_pred>, _pr1,                   \
  int, _op,                              \
  ptr<ast_pred>, _pr2

#define ast_uny_l_pred_fields            \
  int, _op,                              \
  ptr<ast_pred>, _pr

#define ast_literal_expr_fields          \
  fp_t, _vl

#define ast_id_expr_fields               \
  ptr<id>, _id

#define ast_bin_op_expr_fields           \
  ptr<ast_expr>, _ex1,                   \
  int, _op,                              \
  ptr<ast_expr>, _ex2

#define ast_uny_op_expr_fields           \
  int, _op,                              \
  ptr<ast_expr>, _ex

#define ast_fun_call_expr_fields         \
  ptr<id>, _id,                          \
  ptr<list<ptr<ast_expr>>>, _exs

#define ast_syntax_error_fields          \
  string, _str

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

struct symbol_table
{
  void define_fun(ptr<ast_fun_def> _fd);
  void undefine_fun(ptr<ast_fun_def> _fd);
  ptr<ast_fun_def> get_fun_def(ptr<id> _id);

  void open_scope();
  void close_scope();

  void declare_var(ptr<id> _id);
  bool var_is_declared(ptr<id> _id);
```

```
  void set_var(ptr<id> _id, fp_t value);
  maybe_fp_t get_var(ptr<id> _id);
  bool var_is_set(ptr<id> _id);

  map<id,ptr<ast_fun_def>> functions;
  stack<map<id,fp_t>> scopes;
};


/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

struct ast_node
{
  ast_node() {}
  ast_node(ctor_params(ast_node_fields))
    : init_list(ast_node_fields) {}
  virtual ~ast_node() {}

  field_decls(ast_node_fields);
};

struct ast_expr : virtual ast_node
{
  ast_expr() {}
  virtual ~ast_expr() {}
  virtual bool is_valid(symbol_table& sym) = 0;
  virtual fp_t eval(symbol_table& sym) = 0;
  virtual bool is_plottable() { return false; };
};

struct ast_literal_expr : virtual ast_node, ast_expr
{
  ast_literal_expr() {}
  ast_literal_expr(ctor_params(ast_literal_expr_fields), int _ln)
    : init_list(ast_literal_expr_fields), ast_node(_ln) {}
  ~ast_literal_expr() {}
  virtual bool is_valid(symbol_table& sym);
  fp_t eval(symbol_table& sym);

  field_decls(ast_literal_expr_fields);
};

struct ast_id_expr : virtual ast_node, ast_expr
{
  ast_id_expr() {}
  ast_id_expr(ctor_params(ast_id_expr_fields), int _ln)
    : init_list(ast_id_expr_fields), ast_node(_ln) {}
  ~ast_id_expr()  {}
  virtual bool is_valid(symbol_table& sym);
  fp_t eval(symbol_table& sym);

  field_decls(ast_id_expr_fields);
};

struct ast_bin_op_expr : virtual ast_node, ast_expr
{
  ast_bin_op_expr() {}
  ast_bin_op_expr(ctor_params(ast_bin_op_expr_fields), int _ln)
    : init_list(ast_bin_op_expr_fields), ast_node(_ln) {}
  ~ast_bin_op_expr()  {}
  virtual bool is_valid(symbol_table& sym);
  fp_t eval(symbol_table& sym);

  field_decls(ast_bin_op_expr_fields);
};

struct ast_uny_op_expr : virtual ast_node, ast_expr
{
  ast_uny_op_expr() {}
  ast_uny_op_expr(ctor_params(ast_uny_op_expr_fields), int _ln)
    : init_list(ast_uny_op_expr_fields), ast_node(_ln) {}
```

```
  ~ast_uny_op_expr()  {}
  virtual bool is_valid(symbol_table& sym);
  fp_t eval(symbol_table& sym);

  field_decls(ast_uny_op_expr_fields);
};

struct ast_fun_call_expr : virtual ast_node, ast_expr
{
  ast_fun_call_expr() {}
  ast_fun_call_expr(ctor_params(ast_fun_call_expr_fields), int _ln)
    : init_list(ast_fun_call_expr_fields), ast_node(_ln) {}
  ~ast_fun_call_expr() {}
  virtual bool is_valid(symbol_table& sym);
  virtual fp_t eval(symbol_table& sym);
  virtual bool is_plottable() { return true; };

  field_decls(ast_fun_call_expr_fields);
};

struct ast_pred : virtual ast_node
{
  ast_pred() {}
  virtual ~ast_pred() {}
  virtual bool is_valid(symbol_table& sym) = 0;
  virtual bool test(symbol_table& sym) = 0;
};

struct ast_rel_pred : virtual ast_node, ast_pred
{
  ast_rel_pred() {}
  ast_rel_pred(ctor_params(ast_rel_pred_fields), int _ln)
    : init_list(ast_rel_pred_fields), ast_node(_ln) {}
  ~ast_rel_pred() {}
  virtual bool is_valid(symbol_table& sym);
  virtual bool test(symbol_table& sym);

  field_decls(ast_rel_pred_fields);
};

struct ast_bin_l_pred : virtual ast_node, ast_pred
{
  ast_bin_l_pred() {}
  ast_bin_l_pred(ctor_params(ast_bin_l_pred_fields), int _ln)
    : init_list(ast_bin_l_pred_fields), ast_node(_ln) {}
  ~ast_bin_l_pred() {}
  virtual bool is_valid(symbol_table& sym);
  virtual bool test(symbol_table& sym);

  field_decls(ast_bin_l_pred_fields);
};

struct ast_uny_l_pred : virtual ast_node, ast_pred
{
  ast_uny_l_pred() {}
  ast_uny_l_pred(ctor_params(ast_uny_l_pred_fields), int _ln)
    : init_list(ast_uny_l_pred_fields), ast_node(_ln) {}
  ~ast_uny_l_pred() {}
  virtual bool is_valid(symbol_table& sym);
  virtual bool test(symbol_table& sym);

  field_decls(ast_uny_l_pred_fields);
};

struct ast_stmt : virtual ast_node
{
  ast_stmt() {}
  virtual ~ast_stmt() {}
  virtual bool is_valid(symbol_table& sym) = 0;
  virtual maybe_fp_t exec(symbol_table& sym) = 0;
};
```

```
struct ast_block : virtual ast_node
{
  ast_block() {}
  ast_block(ctor_params(ast_block_fields), int _ln)
    : init_list(ast_block_fields), ast_node(_ln) {}
  ~ast_block() {}
  virtual bool is_valid(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);

  field_decls(ast_block_fields);
};

struct ast_var_assign_stmt : virtual ast_node, ast_stmt
{
  ast_var_assign_stmt() {}
  ast_var_assign_stmt(ctor_params(ast_var_assign_stmt_fields), int _ln)
    : init_list(ast_var_assign_stmt_fields), ast_node(_ln) {}
  ~ast_var_assign_stmt()  {}
  virtual bool is_valid(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);

  field_decls(ast_var_assign_stmt_fields);
};

struct ast_if_then_stmt : virtual ast_node, ast_stmt
{
  ast_if_then_stmt() {}
  ast_if_then_stmt(ctor_params(ast_if_then_stmt_fields), int _ln)
    : init_list(ast_if_then_stmt_fields), ast_node(_ln) {}
  ~ast_if_then_stmt() {}
  virtual bool is_valid(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);

  field_decls(ast_if_then_stmt_fields);
};

struct ast_if_then_else_stmt : virtual ast_node, ast_stmt
{
  ast_if_then_else_stmt() {}
  ast_if_then_else_stmt(ctor_params(ast_if_then_else_stmt_fields), int _ln)
    : init_list(ast_if_then_else_stmt_fields), ast_node(_ln) {}
  ~ast_if_then_else_stmt()  {}
  virtual bool is_valid(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);

  field_decls(ast_if_then_else_stmt_fields);
};

struct ast_while_stmt : virtual ast_node, ast_stmt
{
  ast_while_stmt() {}
  ast_while_stmt(ctor_params(ast_while_stmt_fields), int _ln)
    : init_list(ast_while_stmt_fields), ast_node(_ln) {}
  ~ast_while_stmt() {}
  virtual bool is_valid(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);

  field_decls(ast_while_stmt_fields);
};

struct ast_return_stmt : virtual ast_node, ast_stmt
{
  ast_return_stmt() {}
  ast_return_stmt(ctor_params(ast_return_stmt_fields), int _ln)
    : init_list(ast_return_stmt_fields), ast_node(_ln) {}
  ~ast_return_stmt()  {}
  virtual bool is_valid(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);

  field_decls(ast_return_stmt_fields);
```

```
};

struct ast_plot_cmd : virtual ast_node
{
  ast_plot_cmd() {}
  ast_plot_cmd(ctor_params(ast_plot_cmd_fields), int _ln)
    : init_list(ast_plot_cmd_fields), ast_node(_ln) {}
  ~ast_plot_cmd() {}
  virtual bool is_valid(symbol_table& sym);
  virtual void plot(symbol_table& sym);

  field_decls(ast_plot_cmd_fields);
};

struct ast_fun_def : virtual ast_node, enable_shared_from_this<ast_fun_def>
{
  ast_fun_def() {}
  ast_fun_def(ctor_params(ast_fun_def_fields), int _ln)
    : init_list(ast_fun_def_fields), ast_node(_ln) {}
  ~ast_fun_def() {}
  virtual bool is_valid(symbol_table& sym);

  field_decls(ast_fun_def_fields);
};

struct ast_program : virtual ast_node
{
  ast_program() {}
  ast_program(ctor_params(ast_program_fields), int _ln)
    : init_list(ast_program_fields), ast_node(_ln) {}
  ~ast_program() {}
  virtual bool run();

  field_decls(ast_program_fields);
};

struct ast_syntax_error : virtual ast_node, ast_program, ast_fun_def,
  ast_plot_cmd, ast_block, ast_stmt, ast_pred, ast_expr
{
  ast_syntax_error() {}
  ast_syntax_error(ctor_params(ast_syntax_error_fields), int _ln)
    : init_list(ast_syntax_error_fields), ast_node(_ln) {}
  ~ast_syntax_error()  {}
  virtual bool is_valid(symbol_table& sym);
  virtual fp_t eval(symbol_table& sym);
  virtual bool test(symbol_table& sym);
  virtual maybe_fp_t exec(symbol_table& sym);
  virtual void plot(symbol_table& sym);
  virtual bool run();

  field_decls(ast_syntax_error_fields);
};


#endif /* MYLANGA_AST_H */
```

2.7.2 Implementación de clases y estructuras

Finalmente, su implementación.

```
#include <iostream>
#include <cmath>
using namespace std;

#include "mylanga_fp_t.h"
#include "mylanga_ast.h"
#include "mylanga_sem_types.h"
#include "mylanga_error.h"
#define YYSTYPE mylanga_sem_types

#include "parser.hpp"

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

void symbol_table::define_fun(ptr<ast_fun_def> _fd)
{
  functions[*(_fd->_id)] = _fd;
}

void symbol_table::undefine_fun(ptr<ast_fun_def> _fd)
{
  functions.erase(*(_fd->_id));
}

ptr<ast_fun_def> symbol_table::get_fun_def(ptr<id> _id)
{
  return (functions.count(*_id) > 0) ? functions[*_id] : nullptr;
}

void symbol_table::open_scope()
{
  scopes.emplace();
}

void symbol_table::close_scope()
{
  scopes.pop();
}

void symbol_table::declare_var(ptr<id> _id)
{
  set_var(_id, 0.0);
}

bool symbol_table::var_is_declared(ptr<id> _id)
{
  return get_var(_id).is_valid;
}

void symbol_table::set_var(ptr<id> _id, fp_t value)
{
  if (scopes.empty())
  {
    MYLANGA_INTERNAL_ERROR();
  }

  auto& scope = scopes.top();
  scope[*_id] = value;
}

maybe_fp_t symbol_table::get_var(ptr<id> _id)
{
  if (scopes.empty())
  {
    MYLANGA_INTERNAL_ERROR();
  }
```

```
  auto& scope = scopes.top();
  return (scope.count(*_id) > 0) ? maybe_fp_t(scope[*_id]) : maybe_fp_t();
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

fp_t ast_literal_expr::eval(symbol_table& sym)
{
  return _vl;
}

fp_t ast_id_expr::eval(symbol_table& sym)
{
  maybe_fp_t var = sym.get_var(_id);

  if (not var.is_valid)
  {
    // runtime error
    cerr << MYLANGA_RUNTIME_ERROR << "␣|␣" << \
      "Lectura␣de␣la␣variable␣\'" << *_id << "\'␣sin␣haberle␣asignado␣previamente␣un␣valor." <<
          endl;
    MYLANGA_END_ABRUPTLY();
  }

  return var.value;
}

fp_t ast_bin_op_expr::eval(symbol_table& sym)
{
  switch (_op)
  {
    case OP_PLUS: return _ex1->eval(sym) + _ex2->eval(sym);
    case OP_MINUS: return _ex1->eval(sym) - _ex2->eval(sym);
    case OP_MULT: return _ex1->eval(sym) * _ex2->eval(sym);
    case OP_DIV: return _ex1->eval(sym) / _ex2->eval(sym);
    case OP_EXP: return pow(_ex1->eval(sym), _ex2->eval(sym));
  }

  MYLANGA_INTERNAL_ERROR();
}

fp_t ast_uny_op_expr::eval(symbol_table& sym)
{
  switch (_op)
  {
    case OP_MINUS: return - _ex->eval(sym);
  }

  MYLANGA_INTERNAL_ERROR();
}

fp_t ast_fun_call_expr::eval(symbol_table& sym)
{
  list<fp_t> args;
  for (auto _ex : *_exs)
    args.push_back(_ex->eval(sym));

  sym.open_scope();

  ptr<ast_fun_def> _fd = sym.get_fun_def(_id);

  auto arg_it = args.begin();
  for (auto _param_id : *(_fd->_ids))
    sym.set_var(_param_id, *arg_it++);

  maybe_fp_t ret = _fd->_bl->exec(sym);

  sym.close_scope();

  if (not ret.is_valid)
```

```
  {
    // runtime error
    cerr << MYLANGA_RUNTIME_ERROR << "␣|␣" << \
      "La␣función␣\'" << *_id << "\'␣se␣ejecutó␣sin␣retornar␣un␣valor." << endl;
    MYLANGA_END_ABRUPTLY();
  }

  return ret.value;
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

bool ast_rel_pred::test(symbol_table& sym)
{
  switch (_op)
  {
    case REL_LT: return _ex1->eval(sym) < _ex2->eval(sym);
    case REL_LEQ: return _ex1->eval(sym) <= _ex2->eval(sym);
    case REL_EQ: return _ex1->eval(sym) == _ex2->eval(sym);
    case REL_GEQ: return _ex1->eval(sym) >= _ex2->eval(sym);
    case REL_GT: return _ex1->eval(sym) > _ex2->eval(sym);
  }

  MYLANGA_INTERNAL_ERROR();
}

bool ast_bin_l_pred::test(symbol_table& sym)
{
  switch (_op)
  {
    case L_OR: return _pr1->test(sym) or _pr2->test(sym);
    case L_AND: return _pr1->test(sym) and _pr2->test(sym);
  }

  MYLANGA_INTERNAL_ERROR();
}

bool ast_uny_l_pred::test(symbol_table& sym)
{
  switch (_op)
  {
    case L_NOT: return not _pr->test(sym);
  }

  MYLANGA_INTERNAL_ERROR();
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

maybe_fp_t ast_block::exec(symbol_table& sym)
{
  maybe_fp_t ret;
  for (auto _st : *_sts)
  {
    ret = _st->exec(sym);
    if (ret.is_valid) break;
  }

  return ret;
}

maybe_fp_t ast_var_assign_stmt::exec(symbol_table& sym)
{
  sym.set_var(_id, _ex->eval(sym));

  return maybe_fp_t();
}

maybe_fp_t ast_if_then_stmt::exec(symbol_table& sym)
{
  maybe_fp_t ret;
```

```
  if (_pr->test(sym))
    ret = _bl->exec(sym);

  return ret;
}

maybe_fp_t ast_if_then_else_stmt::exec(symbol_table& sym)
{
  maybe_fp_t ret;
  if (_pr->test(sym))
    ret = _bl1->exec(sym);
  else
    ret = _bl2->exec(sym);

  return ret;
}

maybe_fp_t ast_while_stmt::exec(symbol_table& sym)
{
  maybe_fp_t ret;
  while (_pr->test(sym))
  {
    ret = _bl->exec(sym);
    if (ret.is_valid) break;
  }

  return ret;
}

maybe_fp_t ast_return_stmt::exec(symbol_table& sym)
{
  return maybe_fp_t(_ex->eval(sym));
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

void ast_plot_cmd::plot(symbol_table& sym)
{
  sym.open_scope();

  fp_t range_from = _ex1->eval(sym),
  range_step = _ex2->eval(sym),
  range_to = _ex3->eval(sym);

  for (fp_t x = range_from; x <= range_to; x += range_step)
  {
    sym.set_var(_id, x);

    fp_t x_value = _ex_x->eval(sym),
    y_value = _ex_y->eval(sym);

    cout << x_value << "␣" << y_value << endl;

  }

  sym.close_scope();
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

bool ast_literal_expr::is_valid(symbol_table& sym)
{
  return true;
}

bool ast_id_expr::is_valid(symbol_table& sym)
{
  bool res = true;
  if (not sym.var_is_declared(_id))
  {
    cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
```

```
      "La␣variable␣\'" << *_id << "\'␣no␣se␣encuentra␣previamente␣declarada." << endl;
    res = false;
  }

  return res;
}

bool ast_bin_op_expr::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _ex1->is_valid(sym) and res;
  res = _ex2->is_valid(sym) and res;

  return res;
}

bool ast_uny_op_expr::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _ex->is_valid(sym) and res;

  return res;
}

bool ast_fun_call_expr::is_valid(symbol_table& sym)
{
  bool res = true;

  do
  {
    ptr<ast_fun_def> _fd = sym.get_fun_def(_id);
    if (_fd == nullptr)
    {
      cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
        "La␣función␣\'" << *_id << "\'␣no␣se␣encuentra␣definida." << endl;
      res = false; break;
    }

    if (_fd->_ids->size() != _exs->size())
    {
      cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
        "La␣función␣" << (*(_fd->_id)) << "␣recibe␣" << to_string(_fd->_ids->size()) << \
        "␣parámetro(s),␣pero␣es␣invocada␣con␣" <<  to_string(_exs->size()) << \
        "␣argumento(s)." << endl;
      res = false; break;
    }
  } while (false);

  for (auto _ex : *_exs)
    res = _ex->is_valid(sym) and res;

  return res;
}

bool ast_rel_pred::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _ex1->is_valid(sym) and res;
  res = _ex2->is_valid(sym) and res;

  return res;
}

bool ast_bin_l_pred::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _pr1->is_valid(sym) and res;
  res = _pr2->is_valid(sym) and res;

  return res;
}
```

```
bool ast_uny_l_pred::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _pr->is_valid(sym) and res;

  return res;
}

bool ast_block::is_valid(symbol_table& sym)
{
  bool res = true;

  for (auto _st : *_sts)
    res = _st->is_valid(sym) and res;

  return res;
}

bool ast_var_assign_stmt::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _ex->is_valid(sym) and res;

  sym.declare_var(_id);

  return res;
}

bool ast_if_then_stmt::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _pr->is_valid(sym) and res;
  res = _bl->is_valid(sym) and res;

  return res;
}

bool ast_if_then_else_stmt::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _pr->is_valid(sym) and res;
  res = _bl1->is_valid(sym) and res;
  res = _bl2->is_valid(sym) and res;

  return res;
}

bool ast_while_stmt::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _pr->is_valid(sym) and res;
  res = _bl->is_valid(sym) and res;

  return res;
}

bool ast_return_stmt::is_valid(symbol_table& sym)
{
  bool res = true;
  res = _ex->is_valid(sym) and res;

  return res;
}

bool ast_plot_cmd::is_valid(symbol_table& sym)
{
  bool res = true;

  sym.open_scope();
```

```
  do
  {
    res = _ex1->is_valid(sym) and res;
    res = _ex2->is_valid(sym) and res;
    res = _ex3->is_valid(sym) and res;
    if (not res) break;

    fp_t range_from = _ex1->eval(sym),
    range_step = _ex2->eval(sym),
    range_to = _ex3->eval(sym);

    if (not (range_from <= range_to and 0 < range_step))
    {
      cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
        "En␣la␣instrucción␣de␣plot,␣el␣rango␣de␣evaluación␣es␣inválido;␣un␣rango␣válido␣a..d..b
            ␣debe␣cumplir␣a␣<=␣b␣y␣0␣<␣d." << endl;
      res = false;
    }

    if (not (_ex_x->is_plottable() and _ex_y->is_plottable()))
    {
      cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
        "En␣la␣instrucción␣de␣plot,␣las␣expresiones␣a␣evaluar␣deben␣ser␣de␣tipo␣llamado␣a␣
            función." << endl;
      res = false;
    }

    sym.declare_var(_id);

    res = _ex_x->is_valid(sym) and res;
    res = _ex_y->is_valid(sym) and res;

  } while (false);

  sym.close_scope();

  return res;
}

bool has_repeated_elements(ptr<list<ptr<id>>> _ids);
bool ast_fun_def::is_valid(symbol_table& sym)
{
  bool res = true;

  if (sym.get_fun_def(_id) != nullptr)
  {
    cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
      "La␣función␣\'" << *_id << "\'␣ya␣está␣definida." << endl;
    res = false;
  }

  if (has_repeated_elements(_ids))
  {
    cerr << MYLANGA_PARSE_ERROR(_ln) << "␣|␣" << \
      "La␣función␣\'" << *_id << "\'␣contiene␣parámetros␣repetidos␣en␣su␣definición." << endl;
    res = false;
  }

  // para soportar recursión
  sym.define_fun(shared_from_this());
  sym.open_scope();

  for (auto _id : *_ids)
    sym.declare_var(_id);

  res = _bl->is_valid(sym) and res;

  sym.close_scope();
  sym.undefine_fun(shared_from_this());

  return res;
```

```
}

bool has_repeated_elements(ptr<list<ptr<id>>> _ids)
{
  for (auto _id1 : *_ids)
    for (auto _id2 : *_ids)
      if (_id1 != _id2 and *_id1 == *_id2)
        return true;
  return false;
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

bool ast_program::run()
{
  symbol_table sym;

  bool is_valid = true;
  for (auto _fd : *_fds)
  {
    if (not (_fd->is_valid(sym)))
    {
      is_valid = false;
      continue;
    }

    sym.define_fun(_fd);
  }

  is_valid = _pc->is_valid(sym) and is_valid;

  if (is_valid)
    _pc->plot(sym);

  return is_valid;
}

/*  *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   *   */

bool ast_syntax_error::is_valid(symbol_table& sym)
{
  cerr << MYLANGA_SYNTAX_ERROR(_ln) << "␣|␣" << _str << endl;
  return false;
}

fp_t ast_syntax_error::eval(symbol_table& sym)
{
  MYLANGA_INTERNAL_ERROR();
}

bool ast_syntax_error::test(symbol_table& sym)
{
  MYLANGA_INTERNAL_ERROR();
}

maybe_fp_t ast_syntax_error::exec(symbol_table& sym)
{
  MYLANGA_INTERNAL_ERROR();
}

void ast_syntax_error::plot(symbol_table& sym)
{
  MYLANGA_INTERNAL_ERROR();
}

bool ast_syntax_error::run()
{
  MYLANGA_INTERNAL_ERROR();
}
```

3 Ejemplos y resultados

3.1 Códigos correctos y resultados generados

3.1.1 Primer ejemplo: Parábola

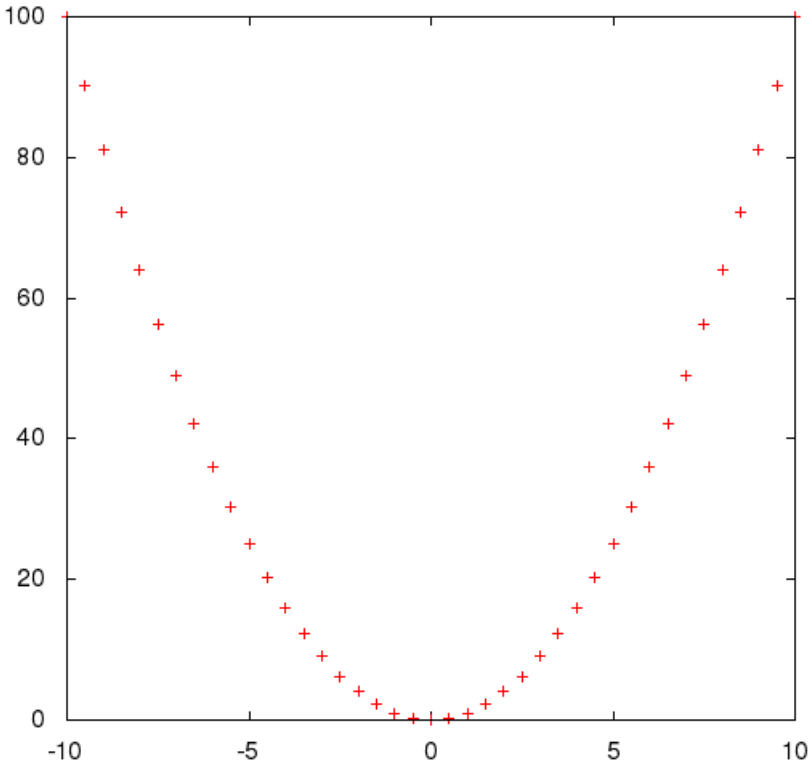
```
function id(x)
  return x

plot (id(x), id(x * x)) for x = -10 .. 0.5 .. 10
```

Este ejemplo simple define la función identidad y en base a eso ploteamos una parábola cuadrática con X entre -10 y 10. Cabe destacar que si bien el ejemplo parece trivial, hace uso de una de las particularidades del lenguaje que es la de definir funciones con cuerpos sin llaves cuando hay una sola instrucción.

El ejemplo devuelve los siguientes puntos esperados:

-10 100
-9.5 90.25
-9 81
-8.5 72.25
...
9.5 90.25
10 100

A su vez graficados obtenemos:



3.1.2 Segundo ejemplo: Seno

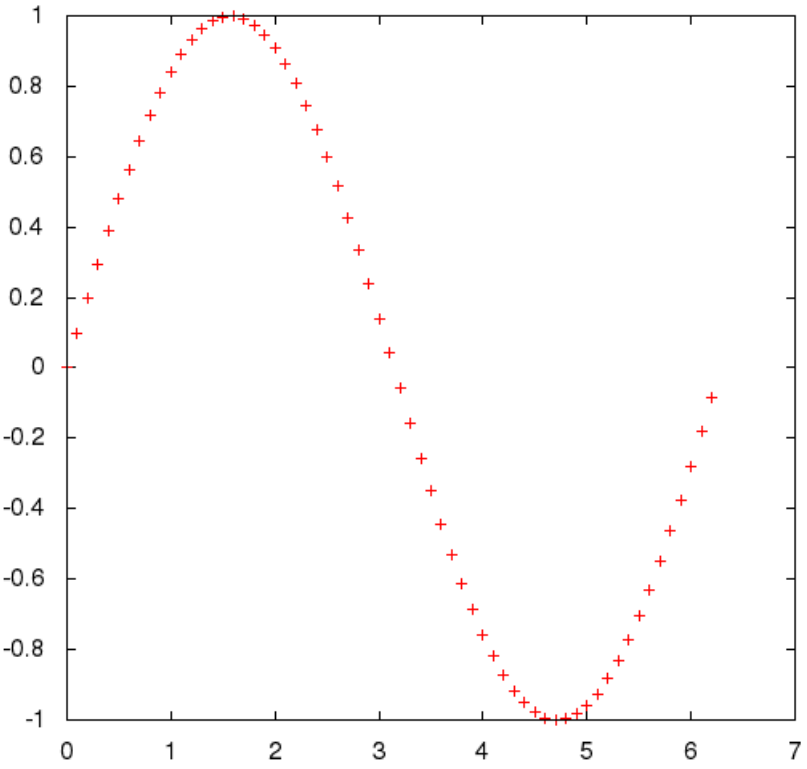
```
function fact(n) {
  if n < 2 then
    return 1
  i = 1
  while n > 0 {
    i = i * n
    n = n - 1
  }
  return i
}

function sin(x) {
  x2 = x * x
  res = 0
  power_x = x
  sign = 1
  i = 0
  while i < 30 {
    res = res + sign * power_x / fact(2 * i + 1)
    power_x = power_x * x2
    sign = - sign
    i = i + 1
  }
  return res
}

function id(x)
  return x

plot (id(x), sin(x)) for x=0..0.1..2*pi
```

Este segundo ejemplo nos deja ver algunas funcionalidades más complejas del lenguaje como las estructuras de control while e if. Podemos observar también que ocurren llamados entre funciones ya que *sin* llama a *fact*. También se observan asignaciones y operaciones aritméticas más complejas. Por último, vemos que se puede llamar a la constante *pi*. Cabe destacar que en este ejemplo hay bloques de instrucciones con y sin llaves en el mismo código.

El ejemplo devuelve los siguientes puntos esperados:

-0.1 0.0998334
-0.2 0.198669
-0.3 0.29552
...
6.2 -0.0830894

A su vez graficados obtenemos la sinusoide esperada:



3.1.3 Tercer ejemplo: Superficie 3D!

```
function fact(n) {
  if n < 2 then
    return 1
  i = 1
  while n > 0 {
    i = i * n
    n = n - 1
  }
  return i
}

function sin(x) {
  while x > 2 * pi {
    x = x - 2 * pi
  }
  x2 = x * x
  res = 0
  power_x = x
  sign = 1
  i = 0
  while i < 30 {
    res = res + sign * power_x / fact(2 * i + 1)
    power_x = power_x * x2
    sign = - sign
```

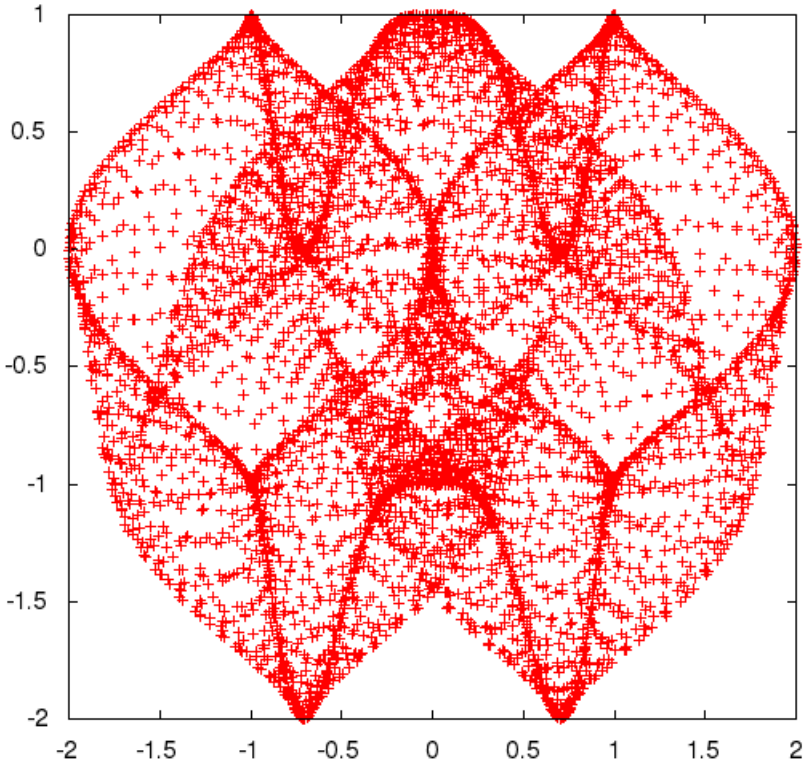
```
    i = i + 1
  }
  return res
}

function cos(x) {
  while x > 2 * pi {
    x = x - 2 * pi
  }
  x2 = x * x
  res = 0
  power_x = 1
  sign = 1
  i = 0
  while i < 30 {
    res = res + sign * power_x / fact(2 * i)
    power_x = power_x * x2
    sign = - sign
    i = i + 1
  }
  return res
}

plot (cos(1 * t) - cos(200 * t) ^ 3, sin(200 * t) - sin(2 * t) ^ 4)
for t = 0..0.001..2*pi
```

Este tercer ejemplo además de ser el más intrincado de los tres propuestos, hace uso de otras dos funcionalidades no vistas en los anteriores. Primero una menor que es la de un *if* con *then* y un bloque de una sola instrucción. Luego, el uso del comando plot con una instrucción en lugar de una función. Como vemos, ambos argumentos de la función plot no son llamados a funciones, sino instrucciones con operaciones aritméticas que a su vez llaman funciones. Esta es una de las modificaciones que propusimos al lenguaje pedido en el enunciado. Si hubiéramos querido hacer

```
cos(1 * t) - cos(200 * t) ^ 3
```

tendríamos que haber hecho una función que hiciera eso y llamarla desde plot. En su lugar, la modificación permite hacerlo inline.

No presentamos los puntos ya que no son de una función conocida y por lo tanto no es simple encontrarle la lógica. Sin embargo, obtenemos el siguiente plot esperado:



3.2 Códigos con errores

Una manera de dar mensajes declarativos a la hora de encontrar un error de sintaxis, es hacer que el parser tenga reglas semánticas para esos errores, y de esa manera pueda dar un mensaje concreto para cada caso. A continuación enumeraremos algunos de los que puede arrojar el parser MyLanga.

3.2.1 Falta de instrucción plot

Si nuestro código solo contiene declaraciones de funciones como en el siguiente ejemplo, sin ningún comando plot al final, obtendremos este mensaje:

```
function id(x)
        return x

# ./mylanga_parser < testError1.my
Error sintáctico detectado en la línea 3. | Falta definir la instrucción de plot.
```

Cabe destacar que esto solo funciona si el código es enviado via pipe. Sino es imposible saber si el bloque de definiciones de funciones ya hay finalizado.

3.2.2 Definición incompleta

En este caso, tenemos definida la función id correctamente y la instrucción plot que llama a id. El error nace de una definición incompleta de la función pepe cuyo bloque de instrucciones está vacío.

```
function id(x)
        return x

function pepe(x)

plot(id(x),id(x)) for x = 1..1..5

# ./mylanga_parser < testError2.my
Error sintáctico detectado en la línea 6. | La definición de la función 'pepe'
está incompleta.
```

3.2.3 Falta de keyword then

En este caso, nos olvidamos de poner el then luego de la guarda del if. Vemos como no solo el error nos indica eso, sino que también el error implica que la función test no se puede definir correctamente, y por lo tanto ambas sentencias del plot fallan con el mismo error.

```
function test(x)
{
        if x==2
                x=1
        return x
}

plot(test(x),test(x)) for x = 1..1..7

# ./mylanga_parser < testError3.my
Error sintáctico detectado en la línea 4. | Falta la palabra clave 'then'
en la instrucción if.
Error de parseo detectado en la línea 8 | La función 'test' no se encuentra definida.
Error de parseo detectado en la línea 8 | La función 'test' no se encuentra definida.
```

Estos son algunos de los errores detectados en el parseo, aunque son muchos otros los que se pueden detectar. Sin embargo, también pueden haber errores durante la etapa de ejecución. A continuación veremos uno.

3.2.4 La función no devuelve un valor

Como vemos en el código, la función solo retorna con x=2.

```
function test(x)
        if x==2 then return x

plot(test(x),test(x)) for x = 1..1..7

# ./mylanga_parser < testError4.my
Error en tiempo de ejecución. | La función 'test' se ejecutó sin retornar un valor.
(x) Terminando la ejecución abruptamente ...
```

Mostramos este ejemplo en particular entre otros, ya que el chequeo de retorno en una función es algo que en otros lenguajes se detecta en la etapa de parseo, aunque con cierta complejidad. Esto implicaría saber si existe alguna rama del árbol que genera el flujo de control de la función que termina sin retornar un valor. En este caso decidimos hacerlo más simple y detectarlo en tiempo de ejecución...

4 Preguntas

4.1 Si queremos que las condiciones sean booleanos solamente cómo podemos verificar esto estáticamente? Cómo incide en la gramática? Cómo lo resuelven otros lenguajes de programación?

Nuestra implementación ya contiene este tipo de chequeo de forma estática en la etapa de parseo. Obviamente esto implicó un cambio particular en la gramática que fue el de separar dos tipos de instrucciones entre predicados y el resto de las expresiones. De esta manera las instrucciones con una guarda, solo matchean si lo que está entre paréntesis es de tipo predicado.

Otra manera es la tradicional de manejar estos errores en lenguajes fuertemente tipados, que es mediante la declaración explícita del tipo de todas las variables y de todas las funciones. De esta manera el compilador sabe (o puede saberlo si se quiere hacer el chequeo) cual es el tipo de la expresión en la guarda. Es el tipo de chequeos que se puede n hacer en lenguajes como C, C++, Pascal, etc.

Otra metodología es la inferencia de tipos. Es decir, que ante la ausencia de hints de tipo de las funciones y las variables, el compilador mediante un algoritmo de inferencia de tipos deduce el tipo más general que puede cumplir con las operaciones de la instrucción. Mediante este tipo de chequeos se puede saber si la evaluación de la expresión puede retornar un booleano o no (en este caso, ya que es lo que nos interesa). Es muy común en los lenguajes funcionales fuertemente tipados. C++ es capaz de hacerlo desde el standard 11.

Existen también otras metodologías híbridas nacidas de los compiladores JIT. En general se tratan de tener compiladas más de una versión de la misma función con todas sus signaturas posibles, lo cual permitiría encontrar la versión que devuelva un booleano en la expresión evaluada.

4.2 Por qué no hacen falta terminadores de sentencia (ej .';') como en C/C++? Expliquen por qué hacen falta en esos lenguajes y por qué no en nuestro caso?

En los lenguajes C y C++, el punto y coma (;) es un terminador de *statements*. El mismo es necesario por múltiples razones. Por un lado, en estos lenguajes es posible yuxtaponer sentencias que se pretende ejecutar en consecuencia, pero que también forman una sentencia válida si se las concatena. Por ejemplo:

```
a = b
*c++
```

podría interpretarse como:

```
a = b * c++
```

Esto no sucede en MyLanga ya que al concatenar sentencias, no se genera otra que sea válida.

Por otro lado, el punto y coma (;) delimita una barrera cronológica a partir de la cual se garantiza que se han ejecutado todos los efectos colaterales de las expresiones evaluadas. Estas barreras se denominan *puntos de secuencia*, o *sequence points*[1]. Nuestro lenguaje no tiene efectos colaterales, con lo cual no son necesarias este tipo de separaciones explícitas.

[1] http://en.wikipedia.org/wiki/Sequence_point

4.3 Si quisiéramos que no importe el orden en que están definidas las funciones dentro del código, cómo lo haríamos? Y para soportar recursión?

En nuestra implementación se mantiene un diccionario de símbolos donde se van agregando las funciones definidas a medida que se va parseando el código. En ese mismo momento se hace un chequeo de validez de una expresión fijándose que los llamados a funciones sean a funciones definidas (es decir que esté en el diccionario). Como esto se hace en el órden de parseo, no nos permite hacer lo preguntado. Para cambiarlo habría que postergar ese chequeo hasta que esté finalizado el parseo de todo el bloque completo de definición de funciones, o simplemente no chequearlo y fallar en tiempo de ejecución.
Para soportar recursión explícita es más simple. Basta con agregar al chequeo que la función esté en el diccionario o que sea la misma que se está definiendo. Si la función finalmente se define correctamente no tendremos ningún problema ya que la función existe, y si no, igualmente generaremos otro error que fue el que causó que no se pueda definir en un primer lugar. En particular la implementación nuestra soporta recursión explícita.