Universidad Católica Argentina

Facultad de Ingeniería y Ciencias Agrarias

Teoría de Lenguajes – Cursada 2019

Examen Final

Profesores: Javier Ouret, Ignacio Parravicini
Fecha: 14/10/2020
Alumno: Héctor Buena Maizón - 151521173

Objetivo del trabajo

El objetivo es desarrollar un compilador de 4 pasadas para un lenguaje sencillo. Este compilador está compuesto por un scanner y un parser los cuales permiten generar el árbol sintáctico, un generador de la tabla de símbolos construye la misma, un analizador semántico que realiza el chequeo de tipos y finalmente un generador de código de 3 direcciones(TAC).

Los pasos que sigue el compilador son:
1) Scanner y parser, los cuales construyen el árbol sintáctico.
2) Análisis semántico construyendo la tabla de símbolos.
3) Análisis semántico que realiza control.
4) Generador de código.

Se desarrolló lo anteriormente mencionado como así también una pequeña maquina virtual donde se prueba el código generado.

Descripción del lenguaje

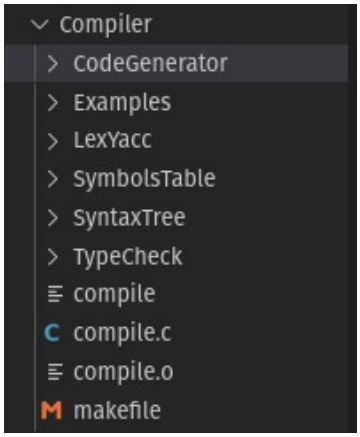
El lenguaje tiene la siguiente sintaxis:

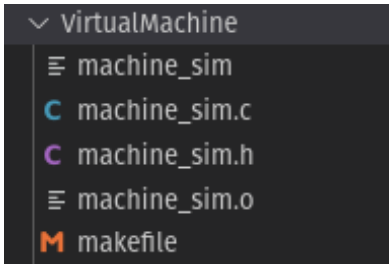
- Un programa es una secuencia de sentencias separadas por ";"
- No tiene procedimientos ni declaraciones
- Las variables utilizadas son todas enteras
- Las variables se declaran al momento de asignarles un valor
- If y repeat se utilizan para control de flujo.
- Se utiliza read y write para control de entrada y salida
- Se evalúan solo expresiones booleanas y aritméticas
- Los comentarios son entre llaves

Símbolos especiales: +, - , *, /, =, <, <=, >=, >, ( ), :=

Componentes del Compilador

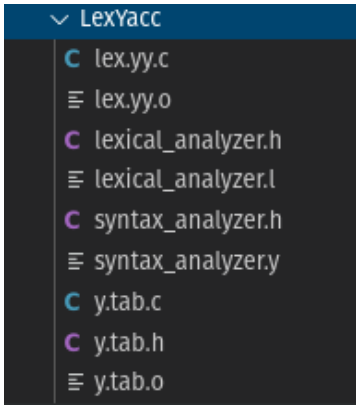
El compilador se compone de las siguientes carpetas:





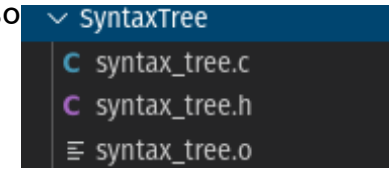
El analizador léxico y el sintáctico, se encuentran dentro de LexYacc, el código del árbol sintáctico se encuentra en SyntaxTree. SymbolsTable, TypeCheck y CodeGenerator tienen las declaración e implementación de la tabla de símbolos, chequeo semántico de tipos y el generador de código para poder ejecutarlo en la virtual Machine.

A continuación se explica en más detalle cual es la función de cada archivo en las carpetas mencionadas:

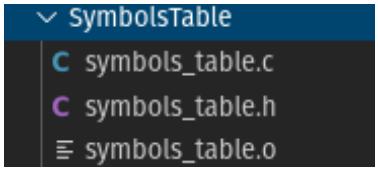


LexYacc:

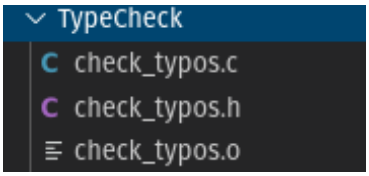
Contiene los archivos “lexical_analyzer.l” y “syntax_analyzer.y”, para el análisis léxico y sintáctico. Además contiene “lexical_analyzer.h” que contiene definición de la función getToken(), como también la definición de las variables para obtener el valor del token y el numero de linea donde se encuentra. “syntax_analyzer.h” contiene los include para el árbol sintáctico.



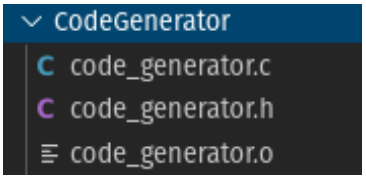
SyntaxTree: “syntax_tree.h” contiene la definición de los NodeType dependiendo si es IF, REPEAT, ASSIGN,READ, WRITE,ID,CONST o OPERATION. En el caso de OperationType se definen también PLUS,SUB,MULT,DIV,EQUAL,LESST,MORET,EQMORET,EQLESST. Por último, se define la estructura de los nodos del árbol y las funciones para crear e imprimir el árbol en pantalla.



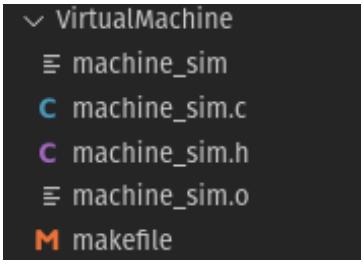
SymbolsTable: En el archivo ‘.h’ se define la estructura que contiene nombre del símbolo, tipo de variable, lineas en las que aparece. Se definen funciones para agregar un símbolo, agregar una linea a un símbolo, mostrar la tabla, construir la tabla a partir del árbol, setear el tipo de variable del símbolo, obtener el tipo de una variable y obtener la posición en memoria de un símbolo. El archivo ‘.c’ contiene la implementación de las funciones mencionadas.



TypeCheck: El archivo ‘.h’ contiene la definición de la función typeCheck, y se definen también unas constantes auxiliares para la función con los tipos de datos INT, BOOL, y NA(no asignado). El archivo ‘.c’ contiene implementación de typeCheck.



CodeGenerator: El archivo ‘.h’ contiene la definición de la función generate code y una estructura auxiliar utilizada para la función. La implementación se encuentra dentro del archivo ‘.c’ .



VirtualMachine: En el archivo ‘.h’ se definen los opcodes, la estructura de las instrucciones, estructuras auxiliares, y la definición de las funciones de la virtual machine. El archivo ‘.c’ contiene la implementación de las funciones.

Todo el proceso de compilación se reúne en “compile.c”, este recibe como parámetro un archivo con el programa a ejecutar deseado, lo compila y guarda un archivo: <filename>.vm .

La maquina virtual en la cual se ejecuta el código compilado está basada en el set de instrucciones RISC y tiene 32 registros, siendo el primero $0 = cero (constante) siguiendo el formato utilizado en MIPS por ejemplo. Además tiene una memoria de datos para enteros y una memoria de instrucciones. Para ejecutar un programa en la misma, se pasa a “machine_sim” el archivo a ejecutar, una vez que carga en memoria el programa, solicita que se presione “y/n” si se desea ejecutar, en el caso de apretar “y” el programa comienza y una vez que finaliza, la maquina virtual también lo hará.

Para poder compilar los archivos del compilador, se debe utilizar el comando make en la carpeta del compiler y luego este mismo comando en la carpeta de la Virtual Machine. Como paso previo, se deben crear los archivos correspondientes al analizador léxico y el sintáctico.
Esto se debe hacer de la siguiente manera:
1) yacc -d syntax_analizer.y
2) lex lexical_analyzer.l

Se utiliza el flag -d de yacc para que se genere el archivo “y.tab.h” que contiene la definición de Tokens.

Análisis Léxico

El primer paso que ejecuta el compilador es el análisis léxico, este toma los lexemes y le retorna los tokens al analizador sintáctico. El archivo lexical analyzer es el siguiente:

```
/*Lexical analyzer*/
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
#include "lexical_analyzer.h"
%}
%option noyywrap
digit   [0-9]
letter  [a-zA-Z]
number {digit}+
identifier {letter}+
newline \n
whitespace [\t ]+


%%
if {return IF;}
then {return THEN;}
else {return ELSE;}
repeat {return REPEAT;}
```

```
until {return UNTIL;}
read {return READ;}
write {return WRITE;}
end {return END;}
"+" {return PLUS;}
"-" {return SUB;}
"*" {return MULT;}
"/" {return DIV;}
"=" {return EQUAL;}
":=" {return ASSIGN;}
"<" {return LESST;}
">" {return MORET;}
">=" {return EQMORET;}
"<=" {return EQLESST;}
";" {return SEMIC;}
"(" {return LPAREN;}
")" {return RPAREN;}
{number} {return NUM;}
{identifier} {return ID;}
{newline} {lineNo++;}
{whitespace} {/*Skip whitespace*/}
"{" { char c;
    do{
        c = input();
        if (c=='\n') {
            lineNo++;
        }
    }while(c!='}' && c!=0);
    if (c==0){
        return ERROR;
    }
    }

%%

void printToken(int aToken){
    switch(aToken){
        // Reserved words
        case IF:
        case ELSE:
        case THEN:
        case REPEAT:
        case UNTIL:
        case READ:
        case WRITE:
        case END:
            printf("Line:%d ,Reserved word:%s\n", lineNo, token_str);
            break;

        // Number or variable identifier
        case NUM:
            printf("Line:%d ,Number:%s\n", lineNo, token_str);
            break;
        case ID:
```

```
            printf("Line:%d ,Identifier:\'%s\'\n", lineNo, token_str);
            break;

        // Special symbols
        case PLUS:
        case SUB:
        case MULT:
        case DIV:
        case EQUAL:
        case EQLESST:
        case EQMORET:
        case ASSIGN:
        case LESST:
        case MORET:
        case SEMIC:
        case RPAREN:
        case LPAREN:
            printf("Line:%d ,symbol:\'%s\'\n", lineNo, token_str);
            break;
    }

}

int getToken(){
    static int firstLine = 1;
    if (firstLine){
      firstLine = 0;
      lineNo = 1;
    }
    int aToken;
    aToken = yylex();
    strncpy(token_str, yytext,TOKENLENGTH);
    printToken(aToken);
    return aToken;
}
```

Análisis Sintáctico

Para el análisis sintáctico “syntax_analyzer.y” participa, en este se define la gramática y los tokens. El compilador llama a la función parse() que una vez finalizado retorna el árbol sintáctico.

```
%{
#include "../SyntaxTree/syntax_tree.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "lexical_analyzer.h"
#define YYSTYPE YYSTYPE
typedef SyntaxTree *YYSTYPE;

static char var_name[TOKENLENGTH];
```

```
static int curLineNo;
int yyerror(char *errmsg);
static SyntaxTree * syntaxTree;
static int yylex(void);
%}

%start program

%token IF THEN ELSE REPEAT UNTIL READ WRITE END
%token PLUS SUB MULT DIV EQUAL LESST MORET EQMORET EQLESST SEMIC LPAREN RPAREN
%token ID NUM ASSIGN
%token ERROR

%%
program : stmt_seq
        {syntaxTree = $1;}
        ;

stmt_seq : stmt_seq stmt
         {
           SyntaxTree * temp = $1;
           if(temp != NULL){
             while (temp->nextStmt != NULL){
               temp = temp->nextStmt;
             }
           temp->nextStmt = $2;
           $$ = $1;
           }
           else {
            $$ = $2;
           }
         }
         | stmt
         {
           $$ = $1;
         }
         ;

stmt : if_stmt { $$ = $1;  }
      | repeat_stmt SEMIC{ $$ = $1; }
      | assign_stmt SEMIC{ $$ = $1; }
      | read_stmt SEMIC{ $$ = $1; }
      | write_stmt SEMIC{ $$ = $1; }
      ;

if_stmt : IF exp THEN stmt_seq END
        { $$ = create_node(IF_TYPE);
          $$->leftChild = $2;
          $$->centerChild = $4;
        }
        | IF exp THEN stmt_seq ELSE stmt_seq END
        {
          $$ = create_node(IF_TYPE);
          $$->leftChild = $2;
```

```
          $$->centerChild = $4;
          $$->rightChild = $6;
        }
        ;

repeat_stmt : REPEAT stmt_seq UNTIL exp
            { $$ = create_node(REPEAT_TYPE);
              $$->leftChild = $2;
              $$->centerChild = $4;
              $$->lineNo = lineNo;
            }
            ;

assign_stmt : ID
                {
              strncpy(var_name, token_str,TOKENLENGTH);
              curLineNo = lineNo;}
                ASSIGN exp
            { $$ = create_node(ASSIGN_TYPE);
                    strncpy($$->str_value, var_name,TOKENLENGTH);
              $$->leftChild = $4;
              $$->lineNo = curLineNo;
            }

read_stmt : READ ID
          { $$ = create_node(READ_TYPE);
            $$->lineNo = lineNo;
            strncpy($$->str_value, token_str,TOKENLENGTH);
          }
          ;

write_stmt : WRITE exp
           { $$ = create_node(WRITE_TYPE);
             $$->lineNo = lineNo;
             $$->leftChild = $2;
           }
           ;

exp : simple_exp LESST simple_exp
    {
      $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = LESST_OP;
      $$->lineNo = lineNo;
    }
    | simple_exp MORET simple_exp
    {
      $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = MORET_OP;
      $$->lineNo = lineNo;
    }
```

```
    | simple_exp EQUAL simple_exp
    {
      $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = EQUAL_OP;
      $$->lineNo = lineNo;
    }
    | simple_exp EQLESST simple_exp
    {
      $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = EQLESST_OP;
      $$->lineNo = lineNo;
    }
    | simple_exp EQMORET simple_exp
    {
      $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = EQMORET_OP;
      $$->lineNo = lineNo;
    }
    | simple_exp
    {
      $$ = $1;
    }
    ;

simple_exp : simple_exp PLUS term
    { $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = PLUS_OP;
      $$->lineNo = lineNo;
    }
    | simple_exp SUB term
    {
      $$ = create_node(OPERATION_TYPE);
      $$->leftChild = $1;
      $$->centerChild = $3;
      $$->opType = SUB_OP;
      $$->lineNo = lineNo;
    }
    | term { $$ = $1; }
    ;

term : term MULT factor
     { $$ = create_node(OPERATION_TYPE);
       $$->leftChild = $1;
       $$->centerChild = $3;
       $$->opType = MULT_OP;
       $$->lineNo = lineNo;
```

```
      }
      | term DIV factor
      { $$ = create_node(OPERATION_TYPE);
        $$->leftChild = $1;
        $$->centerChild = $3;
        $$->opType = DIV_OP;
        $$->lineNo = lineNo;
      }
      | factor { $$ = $1; }
      ;

factor : LPAREN exp RPAREN
        { $$ = $2; }
        | NUM
        { $$ = create_node(CONST_TYPE);
          $$->value = atoi(token_str);
        }
        | ID
        { $$ = create_node(ID_TYPE);
          strncpy($$->str_value, token_str,TOKENLENGTH);
          $$->lineNo = lineNo;
        }
        | error
        {
          $$ = NULL;
        }
        ;
%%

int yyerror(char *errmsg){
  printf("\nSyntax error at line: %d", lineNo);
  if(strlen(token_str)!=0)
    printf(", unexpected token:%s\n", token_str);
  else
    printf("\n");
  exit(EXIT_FAILURE);
  return -1;
}

static int yylex(void){
  return getToken();
}

SyntaxTree * parse(){
  yyparse();
  return syntaxTree;
}
```

Análisis Semántico

El análisis se realiza mediante un recorrido del árbol mediante una función recursiva, al comienzo del análisis la tabla de símbolos tiene los tipos de las variables definidos como “NA”(No asignado) y cada vez que se realiza una asignación, se verifica que el valor a asignar

es un INT y si esto ocurre, se setea el valor de la variable a INT en la tabla. Las constantes siempre retornan INT como su tipo para el chequeo. En el caso de las operaciones aritméticas ambos miembros deben ser enteros. En las comparaciones(<,>,=,<=,>=) se verifica también que ambos miembros sean enteros. Para los read se setea el tipo de la variable donde se guarda el valor como INT y en el caso de write se verifica que el valor a imprimir sea entero. Para las condiciones de los bucles y el if se verifica que la expresión sea booleana.
De encontrarse un error, se indica donde se lo encontró y que diferencia entre el tipo esperado y el actual existe.

```
#include "check_typos.h"

char * typeCheck(SyntaxTree *st){
   char aux[MAXTYPELEN],aux2[MAXTYPELEN];
   char *typo=NULL;
   if(st != NULL){
      switch (st->nodeType) {
         case IF_TYPE:
            strncpy(aux,typeCheck(st->leftChild),MAXTYPELEN);
            if(strcmp(aux,BOOL)!=0){
               printf("Type error in if: %s instead of bool at line:%d\n",aux,st->leftChild->lineNo);
               exit(EXIT_FAILURE);
            }
            typeCheck(st->centerChild);
            typeCheck(st->rightChild);
            typeCheck(st->nextStmt);
            break;

         case REPEAT_TYPE:
            strncpy(aux,typeCheck(st->centerChild),MAXTYPELEN);
            if(strcmp(aux,BOOL)!=0){
               printf("Type error in repeat: %s instead of bool at line:%d\n",aux,st->lineNo);
               exit(EXIT_FAILURE);
            }
            typeCheck(st->leftChild);
            typeCheck(st->nextStmt);
            break;

         case ASSIGN_TYPE:
            strncpy(aux,typeCheck(st->leftChild),MAXTYPELEN);
            if(strcmp(aux,INT)!=0){
               printf("Type error in assignment: %s instead of int at line:%d\n",aux,st->lineNo);
               exit(EXIT_FAILURE);
            }
            if(strcmp(getSymbolVarType(st->str_value),NA)==0){
               setSymbolVarType(st->str_value,aux);
            }
            typeCheck(st->nextStmt);
            break;
```

```
case READ_TYPE:
   setSymbolVarType(st->str_value,INT);
   typeCheck(st->nextStmt);
   break;

case WRITE_TYPE:
   strncpy(aux,typeCheck(st->leftChild),MAXTYPELEN);
   if(strcmp(aux,INT)!=0){
      printf("Type error in write: %s instead of int at line:%d\n",aux,st->lineNo);
      exit(EXIT_FAILURE);
   }
   typeCheck(st->nextStmt);
   break;

case ID_TYPE:
   typo = getSymbolVarType(st->str_value);
   break;

case CONST_TYPE:
   typo = INT;
   break;

case OPERATION_TYPE:
   switch (st->opType) {
      case PLUS_OP:
      case SUB_OP:
      case MULT_OP:
      case DIV_OP:
         strncpy(aux,typeCheck(st->leftChild),MAXTYPELEN);
         strncpy(aux2,typeCheck(st->centerChild),MAXTYPELEN);
         if(strcmp(aux,INT)!=0){
            printf("Type error in operation: %s instead of int at line:%d\n",aux,st->lineNo);
            exit(EXIT_FAILURE);
         }
         else if(strcmp(aux2,INT)!=0){
            printf("Type error in operation: %s instead of int at line:%d\n",aux2,st->lineNo);
            exit(EXIT_FAILURE);
         }
         else{
            typo = INT;
         }
         break;

      case EQUAL_OP:
      case LESST_OP:
      case MORET_OP:
      case EQMORET_OP:
      case EQLESST_OP:
         strncpy(aux,typeCheck(st->leftChild),MAXTYPELEN);
         strncpy(aux2,typeCheck(st->centerChild),MAXTYPELEN);
         if(strcmp(aux,INT)!=0){
            printf("Type error in comparison: %s instead of int at line:%d\n",aux,st->lineNo);
            exit(EXIT_FAILURE);
         }
```

```
                else if(strcmp(aux2,INT)!=0){
                    printf("Type error in comparison: %s instead of int at line:%d\n",aux2,st->lineNo);
                    exit(EXIT_FAILURE);
                }
                else{
                    typo = BOOL;
                }
                break;
            }
            break;

        }
    }
    return typo;
}
```

Generación de Código

La generación de código la realiza una función recursiva que recorre el árbol e inserta en el archivo las instrucciones correspondientes, en el caso de los branch, para no tener que insertar etiquetas y luego sustituirlas en un segundo paso, para mayor sencillez se realiza una ejecución sin agregar al archivo para conocer la posición a la cual se realiza el branch y luego se procede. La función retorna la ultima linea de código que alcanzó y el registro donde almacenó su resultado en el caso que corresponda, el registro que se utiliza aumenta a medida que se necesiten más temporales para realizar el computo de ser necesario. Las instrucciones de la maquina objetivo son de la arquitectura RISC.

Virtual Machine

Consta de los siguientes pasos en su main():

```
int main(int argc, char *argv[]){
    FILE *programFile = NULL;

    if(argc !=2){
        printf("Check args!\n");
        exit(EXIT_FAILURE);
    }
    if((programFile=fopen(argv[1],"r"))==NULL){
        printf("Unable to open file");
        exit(EXIT_FAILURE);
    }

    cleanStart();

    if(loadToMemory(programFile)==-1){
        exit(EXIT_FAILURE);
    }

    printf("Program is ready!, press [y] to run it: ");
```

```
    char run = 'n';
    scanf("%c",&run);
    if(run=='y'){
        runProgram();
        printf("\nProgram Ended, Bye!\n");
    }
    return 0;
}
```

Se recibe como parámetro el programa a ejecutar, si esto ocurre y se pudo abrir el archivo, se realiza una limpieza de la memoria de la maquina para evitar cualquier error, se pone todo a cero y la memoria de instrucciones son todos NOP. Luego se carga a memoria el programa, esto es leer el archivo y volcarlo en memoria con las instrucciones correspondientes. Finalmente si el usuario escribe 'y' ante la pregunta si desea correr el programa la maquina lo ejecuta y al terminar, finaliza.

Programas Ejemplo

En la carpeta Compiler/Examples se incluyeron diferentes programas ejemplo para compilar y ejecutar.

- factorial.tm (Computa el factorial de un numero)
- fermat.tm (Calcula a partir de 3 valores a,b y c, si existen valores de un ‘n’ hasta un n_max determinado que cumplan la ecuación a^n + b^n = c^n)
- fibo.tm (Calcula el n-esimo numero de fibonacci)
- pow.tm (Calcula x^y)

A continuación utilizo fibo.tm para ejemplificar como funciona el compilador.

```
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/Compiler$ ./compile Examples/fibo.tm
```

Programa:

```
fiboO := 0;
fiboI := 1;
read x;
if x = 0 then
    write fiboO;
end
if x = 1 then
    write fiboI;
end
if x >= 2 then
    x := x - 2;
    repeat
        fiboII := fiboO + fiboI;
        fiboO := fiboI;
        fiboI := fiboII;
        x:= x - 1;
    until x < 0;
    write fiboII;
end
```

Parser:

Line:1 ,Identifier:'fiboO'
Line:1 ,symbol:':='
Line:1 ,Number:0
Line:1 ,symbol:';'
Line:2 ,Identifier:'fiboI'
Line:2 ,symbol:':='
Line:2 ,Number:1
Line:2 ,symbol:';'
Line:3 ,Reserved word:read
Line:3 ,Identifier:'x'
Line:3 ,symbol:';'
Line:4 ,Reserved word:if
Line:4 ,Identifier:'x'
Line:4 ,symbol:'='
Line:4 ,Number:0
Line:4 ,Reserved word:then
Line:5 ,Reserved word:write
Line:5 ,Identifier:'fiboO'
Line:5 ,symbol:';'
Line:6 ,Reserved word:end
Line:7 ,Reserved word:if
Line:7 ,Identifier:'x'
Line:7 ,symbol:'='
Line:7 ,Number:1
Line:7 ,Reserved word:then
Line:8 ,Reserved word:write
Line:8 ,Identifier:'fiboI'
Line:8 ,symbol:';'
Line:9 ,Reserved word:end
Line:10 ,Reserved word:if
Line:10 ,Identifier:'x'
Line:10 ,symbol:'>='
Line:10 ,Number:2
Line:10 ,Reserved word:then
Line:11 ,Identifier:'x'
Line:11 ,symbol:':='
Line:11 ,Identifier:'x'
Line:11 ,symbol:'-'
Line:11 ,Number:2
Line:11 ,symbol:';'
Line:12 ,Reserved word:repeat
Line:13 ,Identifier:'fiboII'
Line:13 ,symbol:':='
Line:13 ,Identifier:'fiboO'
Line:13 ,symbol:'+'
Line:13 ,Identifier:'fiboI'
Line:13 ,symbol:';'
Line:14 ,Identifier:'fiboO'
Line:14 ,symbol:':='

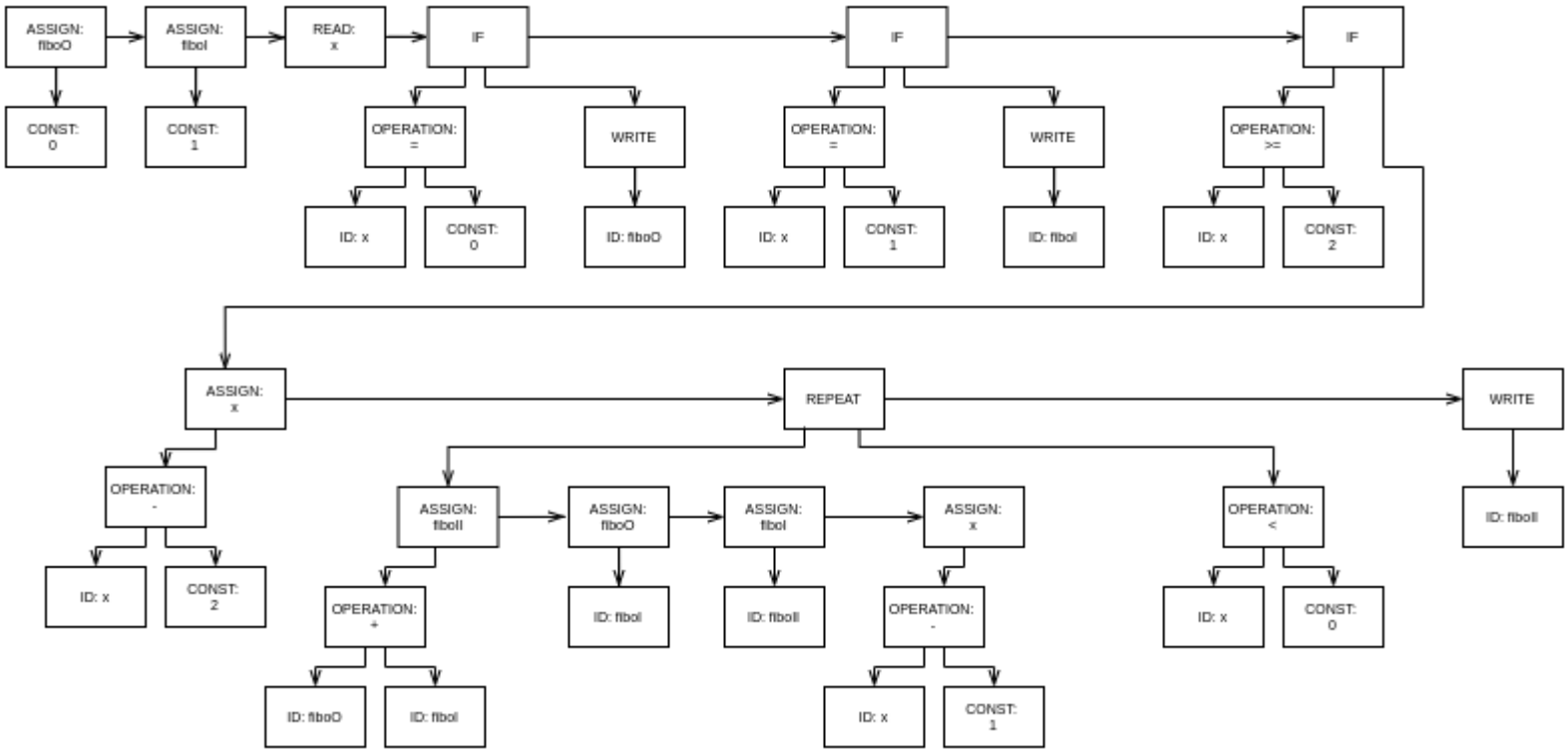
Line:14 ,Identifier:'fiboI'
Line:14 ,symbol:';'
Line:15 ,Identifier:'fiboI'
Line:15 ,symbol:':='
Line:15 ,Identifier:'fiboII'
Line:15 ,symbol:';'
Line:16 ,Identifier:'x'
Line:16 ,symbol:':='
Line:16 ,Identifier:'x'
Line:16 ,symbol:'-'
Line:16 ,Number:1
Line:16 ,symbol:';'
Line:17 ,Reserved word:until
Line:17 ,Identifier:'x'
Line:17 ,symbol:'<'
Line:17 ,Number:0
Line:17 ,symbol:';'
Line:18 ,Reserved word:write
Line:18 ,Identifier:'fiboII'
Line:18 ,symbol:';'
Line:19 ,Reserved word:end

Árbol sintáctico:

Syntax Tree:
ASSIGN: fiboO
LC:
  CONST: 0
NEXT STMT:
ASSIGN: fiboI
LC:
  CONST: 1
NEXT STMT:
READ:x
NEXT STMT:
IF
LC:
  OPERATION: =
  LC:
    ID:x
  CC:
    CONST: 0
CC:
  WRITE:
  LC:
    ID:fiboO
NEXT STMT:
IF
LC:
  OPERATION: =
  LC:
    ID:x
  CC:

```
    CONST: 1
CC:
  WRITE:
  LC:
    ID:fiboI
NEXT STMT:
IF
LC:
  OPERATION: >=
  LC:
    ID:x
  CC:
    CONST: 2
CC:
  ASSIGN: x
  LC:
    OPERATION: -
    LC:
      ID:x
    CC:
      CONST: 2
  NEXT STMT:
  REPEAT
  LC:
    ASSIGN: fiboII
    LC:
      OPERATION: +
      LC:
        ID:fiboO
      CC:
        ID:fiboI
    NEXT STMT:
    ASSIGN: fiboO
    LC:
      ID:fiboI
    NEXT STMT:
    ASSIGN: fiboI
    LC:
      ID:fiboII
    NEXT STMT:
    ASSIGN: x
    LC:
      OPERATION: -
      LC:
        ID:x
      CC:
        CONST: 1
  CC:
    OPERATION: <
    LC:
      ID:x
    CC:
      CONST: 0
  NEXT STMT:
```

WRITE:
LC:
ID:fiboII



Análisis semántico:

```
Checking typos: Typos OK
```

Tabla de Símbolos:

```
Symbols table:
Variable        Lines           Type    Location
fiboO           1,5,13,14,      int     0
fiboI           2,8,13,14,15,   int     1
x               3,4,7,10,11,11,16,16,17,        int     2
fiboII          13,15,18,       int     3
```

Código Generado:

ADDI,1,0,0
SW,1,0
ADDI,1,0,1

```
SW,1,1
READ,1
SW,1,2
LW,2,2
ADDI,3,0,0
SUB,1,2,3
BNE,1,0,12
LW,1,0
WRITE,1
LW,2,2
ADDI,3,0,1
SUB,1,2,3
BNE,1,0,18
LW,1,1
WRITE,1
LW,2,2
ADDI,3,0,2
SUB,1,2,3
BLTZ,1,44
LW,2,2
ADDI,3,0,2
SUB,1,2,3
SW,1,2
LW,2,0
LW,3,1
ADD,1,2,3
SW,1,3
LW,1,1
SW,1,0
LW,1,3
SW,1,1
LW,2,2
ADDI,3,0,1
SUB,1,2,3
SW,1,2
LW,2,2
ADDI,3,0,0
SUB,1,2,3
BGEZ,1,26
LW,1,3
WRITE,1
HALT
```

Ejecución del programa

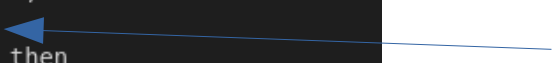
Para mayor facilidad de verificación , los primero 20 números de fibonacci(Comenzando con fibonacci de 0) son: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181.
Algunos ejemplos de ejecución del programa:

```
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/VirtualMachine$ ./machine_sim ../Compiler/Examples/fibo.tm.vm
Program is ready!, press [y] to run it: y
INPUT:8
21

Program Ended, Bye!
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/VirtualMachine$ ./machine_sim ../Compiler/Examples/fibo.tm.vm
Program is ready!, press [y] to run it: y
INPUT:1
1

Program Ended, Bye!
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/VirtualMachine$ ./machine_sim ../Compiler/Examples/fibo.tm.vm
Program is ready!, press [y] to run it: y
INPUT:5
5

Program Ended, Bye!
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/VirtualMachine$ ./machine_sim ../Compiler/Examples/fibo.tm.vm
Program is ready!, press [y] to run it: y
INPUT:3
2

Program Ended, Bye!
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/VirtualMachine$ ./machine_sim ../Compiler/Examples/fibo.tm.vm
Program is ready!, press [y] to run it: y
INPUT:19
4181

Program Ended, Bye!
```

Errores durante la compilación

En el caso de un error, el compilador le anuncia que ocurrió y no genera el archivo con el código ejecutable. Se brinda como ayuda al programador la ubicación del error y en el caso de ser un error de tipos cual fue el tipo incorrecto que se encontró.
Si remuevo un ';' se obtiene el siguiente error:

```
1   fibo0 := 0;
2   fibo1 := 1;
3   read x
4   if x = 0 then
5       write fibo0;
6   end
7   if x = 1 then
8       write fibo1;
9   end
10  if x >= 2 then
11      x := x - 2;
12      repeat
13          fibo2 := fibo0 + fibo1;
14          fibo0 := fibo1;
15          fibo1 := fibo2;
16          x:= x - 1;
17      until x < 0;
18      write fibo2;
19  end
20
```



```
(base) hector@pop-os:~/Desktop/Facultad/Teoria de Lenguajes/MiniCompiler/Compiler$ ./compile Examples/fibo.tm
Line:1 ,Identifier:'fibo0'
Line:1 ,symbol:':='
Line:1 ,Number:0
Line:1 ,symbol:';'
Line:2 ,Identifier:'fiboI'
Line:2 ,symbol:':='
Line:2 ,Number:1
Line:2 ,symbol:';'
Line:3 ,Reserved word:read
Line:3 ,Identifier:'x'
Line:4 ,Reserved word:if

Syntax error at line: 4, unexpected token:if
```

Si en cambio provoco un error de tipos, colocando un entero en lugar de un boolean:

```
1   fibo0 := 0;
2   fiboI := 1;
3   read x;
4   if x = 0 then
5       write fibo0;
6   end
7   if x = 1 then
8       write fiboI;
9   end
10  if x >= 2 then
11      x := x - 2;
12      repeat
13          fiboII := fibo0 + fiboI;
14          fibo0 := fiboI;
15          fiboI := fiboII;
16          x:= x - 1;
17      until 0 ;
18      write fiboII;
19  end
20
```

Se sustituyo ‘x < 0 ‘ por el entero 0.

```
Checking typos: Type error in repeat: int instead of bool at line:17
```