Universidad de Guanajuato

Campus Irapuato-Salamanca

Licenciatura en Ingeniería en Sistemas Computacionales

Compiladores

Proyecto Final: Compilador de lenguaje C

Alumnos:

Hernández Baca Alejandro (790684)

Aguirre Salinas Giovanni Daniel (146804)

Ramos Andrade Samuel Jesús (146878)

Profesor: Dr. José Ruiz Pinales

Fecha: 13/12/2020

Introducción:

En el presente documento se muestra el proceso que se llevó a cabo para la creación de un compilador para lenguaje C. Todo fue realizado en equipo y se hicieron pruebas por separado, para tener la seguridad de que funciona de manera correcta.

Desarrollo:

A continuación, se muestra el código completo del compilador, siendo este llamado “ansic85.y”, el cual se describen las reglas necesarias para poder compilar pequeños programas en C, los cuales pueden llevar desde un simple printf y declaración de variables y funciones, hasta un ciclo for, while o do while.

A lo largo del código se muestran pequeños comentarios encerrados en los símbolos “/* *comentario* */” para una mejor explicación de cada una de las funciones del mismo.

Al correr el ejecutable y mandarle un archivo .c para compilar, este mostrará en pantalla el código intermedio y generará el correspondiente archivo en lenguaje ensamblador.

```
%{

#include <stdio.h>

#include<stdlib.h>

#include<iostream>

#include<iomanip>

#include<fstream>

#include<string>

#include<cstring>

#include<list>


using namespace std;


extern int yylex(void);

extern void yyerror(const char *);

extern FILE *yyin;


/*Definir los tipos de variables*/

enum VARTYPES{

VOIDVAL = 1,
```

```
CHARVAL,
UCHARVAL,
SCHARVAL,
SHORTVAL,
USHORTVAL,
INTVAL,
UINTVAL,
LONGVAL,
ULONGVAL,
FVAL,
DOUBLEVAL,
LDOUBLEVAL,
STRVAL,
STRUCVAL,
UNIONVAL,
ENUMVAL,
TYPENM
};


struct symrec{
std::string name; /*name of symbol*/
int size; /*Indicar el tamano en bytes que ocupa de espacio la variable*/
int init; /*si el simbolo es inicializado*/
int typ; /*indicar tipo de la variable o funcion*/
int offset; /*offset en bloque de datos donde estan los datos de la variable*/
struct{ /*Guardar informacion sobre argumentos de entrada de una funcion*/
int func_type; /*Indicar el tipo de valor de retorno*/
std::list<struct symrec *> *sym_table;
} func_desc;
};
```

```
typedef struct symrec symrec;

typedef std::list<symrec *> symboltable;

symboltable *sym_table = NULL;

/*Tabla de variables locales (para bloque entre {})*/
symboltable *localsyms = NULL;
/*Tabla de variables locales de funcion siendo analizada*/
symrec *curr_func = NULL;

/*Implementar stack de ambitos de variables como lista (tiene metodos push y pop)*/
std::list<std::list<symrec *> *> scopes;
/*
0 ambito de variables globales
1 indica ambito de variables locales de funcion
2 indica ambito de variables locales de bloque
3 indica ambito de variables locales de bloque anidado
...
*/
int currscope = 0; /*Para saber en que ambito estamos*/

int data_offset = 0; /*offset inicial en bloque de dattos*/
unsigned char datablock[64*1024]; /*Bloque de datos de 64 KB*/

typedef union VALUE{
```

```
char charval;
int intval;
float floatval;
double doubleval;
} VALUE;

typedef struct CONST_DATA{
VALUE val;
int typ;
} CONST_DATA;

typedef struct CASE_DATA{
VALUE val;
int typ;
int addr;
} CASE_DATA;

typedef struct VAR_DATA{
std::string *name;
VALUE val;
int init;
int typ;
int op;
symrec *var;
symrec *var2;
std::list<int> *truelist;
std::list<int> *falselist;
std::list<VAR_DATA *> *arglist;
} VAR_DATA;
```

```
enum TYPE_SPECIFIERS{
VOID_SPEC = 0x0000001,
CHAR_SPEC = 0x0000002,
SHORT_SPEC = 0x0000004,
INT_SPEC = 0x0000005,
LONG_SPEC = 0x0000010,
FLOAT_SPEC = 0x0000020,
DOUBLE_SPEC = 0x0000040,
STRING_SPEC = 0x0000080,
STRUCT_SPEC = 0x0000100,
UNION_SPEC = 0x0000200,
ENUM_SPEC = 0x0000400,
TYPENAME_SPEC = 0x0000800,
VAR_SPEC = 0x0001000,
ARRAY_SPEC = 0x0002000,
FUNC_SPEC = 0x0004000,
SIGNED_SPEC = 0x0008000,
UNSIGNED_SPEC = 0x0010000,
CONST_SPEC = 0x0020000,
VOLATILE_SPEC = 0x0040000,
STATIC_SPEC = 0x0080000,
TYPEDEF_SPEC = 0x0100000,
EXTERN_SPEC = 0x0200000,
AUTO_SPEC = 0x0400000,
REGISTER_SPEC = 0x0800000,
POINTER_SPEC = 0x1000000,
ARG_SPEC = 0x2000000,
ARGELLIP_SPEC = 0x4000000,
};
enum ASSIGN_TYPES{
```

```
EQ_ASSIGN_OP = 1,
MUL_ASSIGN_OP,
DIV_ASSIGN_OP,
MOD_ASSIGN_OP,
ADD_ASSIGN_OP,
SUB_ASSIGN_OP,
LEFT_ASSIGN_OP,
RIGHT_ASSIGN_OP,
AND_ASSIGN_OP,
XOR_ASSIGN_OP,
OR_ASSIGN_OP
};

enum OP_TYPES{
PLUS_OP = 1,
ADDR_OP,
MINUS_OP,
DEREF_OP,
TWOCOMP_OP,
NOT_OP
};

/*Codigos de representacion intermedia*/
enum code_ops{ STORE_IR, STOREA_IR, LOADA_IR, IF_EQ_IR, IF_NE_IR, IF_LT_IR,
IF_GT_IR, IF_LE_IR, IF_GE_IR, GOTO_IR,
ADD_IR, SUB_IR, MULT_IR, DIV_IR,
MINUS_IR, MOD_IR, INC_IR, DEC_IR, ADDRESS_IR,
DEREF_IR, TWOCOMP_IR, NOT_IR, INT_IR, FLOAT_IR, CHAR_IR,
LSHIFT_IR, RSHIFT_IR, AND_IR, XOR_IR, OR_IR, RET_IR, PROC_IR, ENDPROC_IR,
CALL_IR, PARAM_IR
};
```

```
typedef struct {
enum code_ops op;
symrec *arg1;
symrec *arg2;
symrec *result;
} quad;

/*Codigo intermedio*/
quad code[9999];
int nextinstr = 0; /*Initial offset in code array*/

struct labrec{
std::string name; /*Name of label*/
int instr; /*Instruction offset*/
};
std::list<labrec *> lab_table;

#include "symdef.h"
#include "genlib.h"
%}

%union{
/*struct {char cval;int ival;double dval;int typ;} val;*/
struct{
union{
char cval;
```

```
char *str;
char *name;
int ival;
float fval;
double dval;
};
int type;
} token;
VAR_DATA *id_data;
std::list<VAR_DATA *> *idlist;
int scsp;
int qual;
int asop;
int instr;
int typ;
int op;
std::list<int> *sqlist;
VAR_DATA sym; /*para poner datos sobre una variable ya declarada*/
struct{
std::list<int> *breaklist;
std::list<int> *nextlist;
std::list<CASE_DATA *> *caselist;
} lists;
}


%token <token.name>IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME
```

```
%token TYPEDEF EXTERN STATIC AUTO REGISTER

%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE
VOID

%token STRUCT UNION ENUM ELLIPSIS


%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK
RETURN


%nonassoc NO_ELSE

%nonassoc ELSE


%type <id_data> direct_declarator

%type <id_data> init_declarator

%type <id_data> declarator

%type <idlist> init_declarator_list

%type <sqlist> declaration_specifiers

%type <scsp> storage_class_specifier

%type <token.type> type_specifier

%type <qual> type_qualifier

%type <sym> primary_expression

%type <sym> postfix_expression

%type <sym> unary_expression

%type <sym> cast_expression

%type <sym> multiplicative_expression

%type <sym> additive_expression

%type <sym> shift_expression

%type <sym> relational_expression

%type <sym> equality_expression
```

```
%type <sym> and_expression
%type <sym> exclusive_or_expression
%type <sym> inclusive_or_expression
%type <sym> logical_and_expression
%type <sym> logical_or_expression
%type <sym> conditional_expression
%type <sym> assignment_expression
%type <sym> expression
%type <sym> initializer
%type <asop> assignment_operator
%type <instr> N
%type <lists> selection_statement
%type <lists> statement
%type <lists> labeled_statement
%type <lists> compound_statement
%type <lists> iteration_statement
%type <lists> statement_list
%type <lists> jump_statement
%type <sym> argument_expression_list
%type <lists> M
%type <sym> parameter_declaration
%type <idlist> parameter_list
%type <idlist> parameter_type_list
%type <sym> logic_expression
%type <sym> expression_statement
%type <sym> constant_expression
%type <op> unary_operator
%type <typ> type_name
%type <sqlist> specifier_qualifier_list


%start translation_unit
```

```
%locations
%%


primary_expression
: IDENTIFIER {
printf("primary_expression : IDENTIFIER\n");
$$.name = new string($1); /*La cadena string que contiene el nombre de la variable*/
symrec *s = getsym($$.name);
if(s == NULL){
printf("Variable %s", $1);
yyerror("not declared!");
}
$$.var = s; /*La variavle de la tabla de simbolos que se encontro con getsym()*/
$$.typ = s->typ; /*El tipo se obtuvo de la tabla de simbolos*/
}
| CONSTANT { printf("primary_expression : CONSTANT\n");
symrec *s;
std::string *name;
switch($<token.type>1){
case INTVAL:
name = newtemp(); /*crear una variable temporal*/
s = putsym(name, INTVAL | CONST_SPEC);
$$.val.intval = $<token.ival>1;
*(int *)(datablock+s->offset) = $<token.ival>1;
$$.typ = s->typ;
$$.var = s;
break;
case DOUBLEVAL:
name = newtemp(); /*crear una variable temporal*/
s = putsym(name, DOUBLEVAL | CONST_SPEC);
$$.val.doubleval = $<token.dval>1;
```

```
*(double *)(datablock+s->offset) = $<token.ival>1;
$$.typ = s->typ;
$$.var = s;
break;
case CHARVAL:
name = newtemp(); /*crear una variable temporal*/
s = putsym(name, CHARVAL | CONST_SPEC);
$$.val.charval = $<token.cval>1;
*(char *)(datablock+s->offset) = $<token.ival>1;
$$.typ = s->typ;
$$.var = s;
break;
}
}
| STRING_LITERAL { printf("primary_expression : STRING_LITERAL\n");
/*Crear variable temporal para guardar cadena*/
std::string *name = newtemp();
symrec *s;
/*Agregar variable temporal a tabla de simbolos*/
s = putsym(name, STRVAL | CONST_SPEC);
/*Determinar tamano de cadena*/
s->size = strlen($1)+1;
/*Copiar cadena a bloque de datos datablock*/
memcpy(datablock+s->offset, $1, s->size);
/*Incrementar valor de posicion libre en bloque de datos*/
data_location(s->size);
/*Propagar atributos de variable temporal al nodo padre primary_expression*/
$$.name =name;
$$.typ = s->typ;
$$.var = s;
}
```

```
| '(' expression ')' {$$ = $2;}
;


postfix_expression
: primary_expression {$$ = $1;}
| postfix_expression '[' expression ']' {}
| postfix_expression '(' ')' {
symrec *s;
std::string *name = newtemp();
s = putsym(name, $1.typ & 0x1F);
$$.typ = s->typ;
gencode(CALL_IR, $1.var, nullptr, s);
$$.var = s;
}
| postfix_expression '(' argument_expression_list ')' {
symrec *s, *s2, *s3;
std::list<VAR_DATA *>::reverse_iterator it;
std::string *name = newtemp();
s = putsym(name, $1.typ & 0x1F);
$$.typ = s->typ;
for(it = $3.arglist->rbegin(); it != $3.arglist->rend(); it++){
gencode(PARAM_IR, (*it)->var, nullptr, nullptr);
}
s3 = new symrec();
s3->name = "";
s3->offset = $3.arglist->size();
gencode(CALL_IR, $1.var, s3, s);
$$.var = s;
}
| postfix_expression '.' IDENTIFIER {}
| postfix_expression PTR_OP IDENTIFIER {}
```

```
| postfix_expression INC_OP {
if($1.typ & VAR_SPEC){
/*Crear variable temporal para guardar valor de postfix_expression antes de incremento*/
std::string *name = newtemp();
symrec *s;
/*Agregar variable temporal a tabla de simbolos*/
s = putsym(name, $1.typ);
/*Generar instruccion de codigo intermedio para asignacion*/
gencode(STORE_IR, $1.var, nullptr, s);
gencode(INC_IR, $1.var, nullptr, $1.var);
$$.name = name;
$$.typ = $1.typ;
$$.var = s;
}
else
yyerror("Error : Operand of increment operator must be an variable name\n");
}
| postfix_expression DEC_OP {
if($1.typ & VAR_SPEC){
/*Crear variable temporal para guardar valor de postfix_expression antes de incremento*/
std::string *name = newtemp();
symrec *s;
/*Agregar variable temporal a tabla de simbolos*/
s = putsym(name, $1.typ);
/*Generar instruccion de codigo intermedio para asignacion*/
gencode(STORE_IR, $1.var, nullptr, s);
gencode(DEC_IR, $1.var, nullptr, $1.var);
$$.name = name;
$$.typ = $1.typ;
$$.var = s;
}
```

```
else
yyerror("Error : Operand of decrement operator must be an variable name\n");
}
;


argument_expression_list
: assignment_expression {
std::list<VAR_DATA *> *newlist = new std::list<VAR_DATA *>();
VAR_DATA *newstruc = new VAR_DATA();
*newstruc = $1;
newlist->push_back(newstruc);
$$.name = new std::string();
$$.typ = 0;
$$.arglist = newlist;
}
| argument_expression_list ',' assignment_expression {
VAR_DATA *newstruc = new VAR_DATA();
*newstruc = $3;
$$ = $1;
$$.arglist->push_back(newstruc);
}
;


unary_expression
: postfix_expression {$$ = $1;}
| INC_OP unary_expression {
if($2.typ & VAR_SPEC){
/*Generar instruccion de codigo intermedio para incremento*/
gencode(INC_IR, $2.var, nullptr, $2.var);
/*Propagar atributos de variable temporal al nodo padre postfix_expression*/
```

```
$$.name = $2.name;
$$.typ = $2.typ;
$$.var = $2.var;
}
else
yyerror("Error : Operand of increment operator must be an variable name\n");
}
| DEC_OP unary_expression {
if($2.typ & VAR_SPEC){
/*Generar instruccion de codigo intermedio para incremento*/
gencode(DEC_IR, $2.var, nullptr, $2.var);
/*Propagar atributos de variable temporal al nodo padre postfix_expression*/
$$.name = $2.name;
$$.typ = $2.typ;
$$.var = $2.var;
}
else
yyerror("Error : Operand of decrement operator must be an variable name\n");
}
| unary_operator cast_expression { printf("unary_expression : unary_operator
cast_expression\n");
std::string *name;
symrec *s;
if($1 == PLUS_OP){
$$ = $2;
}
name = newtemp();
/*Generar instruccion de codigo intermedio*/
switch($1){
case ADDR_OP:
s = putsym(name, INTVAL);
gencode(ADDRESS_IR, $2.var, nullptr, s);
```

```
$$.typ = INTVAL;
break;
case DEREF_OP:
if(!ispointer($2.typ))
yyerror("Error: Operand must be a ponter!\n");
s = putsym(name, $2.typ & ~POINTER_SPEC);
gencode(DEREF_IR, $2.var, nullptr, s);
$$.typ = $2.typ;
break;
case MINUS_OP:
s = putsym(name, $2.typ);
gencode(MINUS_IR, $2.var, nullptr, s);
$$.typ = $2.typ;
break;
case TWOCOMP_OP:
s = putsym(name, INTVAL);
gencode(TWOCOMP_IR, $2.var, nullptr, s);
$$.typ = INTVAL;
break;
case NOT_OP:
s = putsym(name, INTVAL);
gencode(NOT_IR, $2.var, nullptr, s);
$$.truelist = $2.falselist;
$$.falselist = $2.truelist;
$$.typ = INTVAL;
break;
}
$$.name = name;
$$.var = s;
}
| SIZEOF unary_expression {
```

```
printf("unary_expression : SIZEOF unary_expression\n");
/*Crear variable temporal para guardar valor tamano de unary_expression*/
std::string *name = newtemp();
symrec *s;
/*Agregar variable temporal a tabla de simbolos*/
s = putsym(name, INTVAL | CONST_SPEC);
/*Copiar tamano de unary_expression en bloque de datos datablock*/
*(int *)(datablock+s->offset) = getsizeof($2.typ);
/*Propagar atributos de postfix_expression al nodo padre unary_expression*/
$$.name = name;
$$.typ = INTVAL | CONST_SPEC;
$$.var = s;
}
| SIZEOF '(' type_name ')' {
printf("unary_expression : SIZEOF '(' type_name ')'\n");
/*Crear variable temporal para guardar valor tamano de unary_expression*/
std::string *name = newtemp();
symrec *s;
/*Agregar variable temporal a tabla de simbolos*/
s = putsym(name, INTVAL | CONST_SPEC);
/*Copiar tamano de unary_expression en bloque de datos datablock*/
*(int *)(datablock+s->offset) = getsizeof($3);
/*Propagar atributos de postfix_expression al nodo padre unary_expression*/
$$.name = name;
$$.typ = INTVAL | CONST_SPEC;
$$.var = s;
}
;


unary_operator
: '&' {$$ = ADDR_OP;}
```

```
| '*' {$$ = DEREF_OP;}
| '+' {$$ = PLUS_OP;}
| '-' {$$ = MINUS_OP;}
| '~' {$$ = TWOCOMP_OP;}
| '!' {$$ = NOT_OP;}
;


cast_expression
: unary_expression {}
| '(' type_name ')' cast_expression {}
;


multiplicative_expression
: cast_expression {$$ = $1;}
| multiplicative_expression '*' cast_expression { std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(MULT_IR, $1.var, $3.var, s);
$$.name = name;
$$.var = s;
}
| multiplicative_expression '/' cast_expression { std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(DIV_IR, $1.var, $3.var, s);
$$.name = name;
$$.var = s;
}
```

```
| multiplicative_expression '%' cast_expression { std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(MOD_IR, $1.var, $3.var, s);
$$.name = name;
$$.var = s;
}
;

additive_expression
: multiplicative_expression {$$ = $1;}
| additive_expression '+' multiplicative_expression {
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(ADD_IR, $1.var, $3.var, s);
$$.name = name;
$$.var = s;
}
| additive_expression '-' multiplicative_expression { std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(SUB_IR, $1.var, $3.var, s);
$$.name = name;
$$.var = s;
}
;

shift_expression
```

```
: additive_expression {$$ = $1;}
| shift_expression LEFT_OP additive_expression {}
| shift_expression RIGHT_OP additive_expression {}
;


relational_expression
: shift_expression {$$ = $1;}
| relational_expression '<' shift_expression {
symrec *s1 = $1.var, *s2 = $3.var;
$$.truelist = makelist(nextinstr);
gencode(IF_LT_IR, s1, s2, nullptr);
$$.falselist = makelist(nextinstr);
gencode(GOTO_IR);
}
| relational_expression '>' shift_expression {
symrec *s1 = $1.var, *s2 = $3.var;
$$.truelist = makelist(nextinstr);
gencode(IF_GT_IR, s1, s2, nullptr);
$$.falselist = makelist(nextinstr);
gencode(GOTO_IR);
}
| relational_expression LE_OP shift_expression {
symrec *s1 = $1.var, *s2 = $3.var;
$$.truelist = makelist(nextinstr);
gencode(IF_LE_IR, s1, s2, nullptr);
$$.falselist = makelist(nextinstr);
gencode(GOTO_IR);
}
| relational_expression GE_OP shift_expression {
symrec *s1 = $1.var, *s2 = $3.var;
$$.truelist = makelist(nextinstr);
```

```
gencode(IF_LT_IR, s1, s2, nullptr);
$$.falselist = makelist(nextinstr);
gencode(GOTO_IR);
}
;

N: /*Empty*/ {/*Retornar direccion de siguiente intruccion de IR*/
$$ = nextinstr;
}
;

equality_expression
: relational_expression {$$ = $1;}
| equality_expression EQ_OP relational_expression {}
| equality_expression NE_OP relational_expression {}
;

and_expression
: equality_expression {$$ = $1;}
| and_expression '&' equality_expression {}
;

exclusive_or_expression
: and_expression {$$ = $1;}
| exclusive_or_expression '^' and_expression {}
;

inclusive_or_expression
```

```
: exclusive_or_expression {$$ = $1;}
| inclusive_or_expression '|' exclusive_or_expression {}
;


logical_and_expression
: inclusive_or_expression {$$ = $1;}
| logical_and_expression AND_OP N inclusive_or_expression {
backpatch($1.truelist, $3);
$$.truelist = $4.truelist;
$$.falselist = $1.falselist;
$$.falselist->merge(*($4.falselist));
$$.typ = INTVAL;
}
;


logical_or_expression
: logical_and_expression {$$ = $1;}
| logical_or_expression OR_OP N logical_and_expression {
backpatch($1.falselist, $3);
$$.falselist = $4.falselist;
$$.truelist = $1.truelist;
$$.truelist->merge(*($4.truelist));
$$.typ = INTVAL;
}
;


conditional_expression
: logical_or_expression {$$ = $1;}
| logical_or_expression '?' expression ':' conditional_expression {}
```

```
;



assignment_expression
: conditional_expression { printf("assignment_expression : conditional_expression\n");
$$ = $1;
}
| unary_expression assignment_operator assignment_expression {
printf("assignment_expression : unary_expression assignment_operator
assignment_expression\n");
/*Verifica si hat saltos pendientes en assignment_expression
Si los hay, asigna el valor logico 0 o 1*/
if($3.truelist || $3.falselist){
std::string *name = newtemp(); /*Nueva variable tempotal*/
symrec *s1, *s0;
s0 = putsym(name, INTVAL | CONST_SPEC);
*(int*)(datablock+s0->offset) = 0;
name = newtemp();
s1 = putsym(name, INTVAL | CONST_SPEC);
*(int*)(datablock+s1->offset) = 1;
name = newtemp();
$3.var = putsym(name, INTVAL | VAR_SPEC);
$3.typ = INTVAL | VAR_SPEC;
$3.name = name;
backpatch($3.truelist, nextinstr);
gencode(STORE_IR, s1, nullptr, $3.var);
gencode(GOTO_IR, nullptr, nullptr, nextinstr+2);
backpatch($3.falselist, nextinstr);
gencode(STORE_IR, s0, nullptr, $3.var);
}
switch($2){
case EQ_ASSIGN_OP:
```

```
if(!isarray($1.typ))
gencode(STORE_IR, $3.var, nullptr, $1.var);
break;
case MUL_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(MULT_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case DIV_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(DIV_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case MOD_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(MOD_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case ADD_ASSIGN_OP:
```

```
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(ADD_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case SUB_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(SUB_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case LEFT_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(LSHIFT_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case RIGHT_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
```

```
gencode(RSHIFT_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case AND_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(AND_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case XOR_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(XOR_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
case OR_ASSIGN_OP:
if(!isarray($1.typ)){
std::string *name = newtemp(); /*Crear nueva variable temporal para resultado*/
symrec *s;
s = putsym(name, gettype($1.typ, $3.typ) & ~CONST_SPEC | VAR_SPEC);
gencode(OR_IR, $1.var, $3.var, s);
gencode(STORE_IR, s, nullptr, $1.var);
}
break;
```

```
}
}
;

assignment_operator
: '=' {$$ = EQ_ASSIGN_OP;}
| MUL_ASSIGN {$$ = MUL_ASSIGN_OP;}
| DIV_ASSIGN {$$ = DIV_ASSIGN_OP;}
| MOD_ASSIGN {$$ = MOD_ASSIGN_OP;}
| ADD_ASSIGN {$$ = ADD_ASSIGN_OP;}
| SUB_ASSIGN {$$ = SUB_ASSIGN_OP;}
| LEFT_ASSIGN {$$ = LEFT_ASSIGN_OP;}
| RIGHT_ASSIGN {$$ = RIGHT_ASSIGN_OP;}
| AND_ASSIGN {$$ = AND_ASSIGN_OP;}
| XOR_ASSIGN {$$ = XOR_ASSIGN_OP;}
| OR_ASSIGN {$$ = OR_ASSIGN_OP;}
;

expression
: assignment_expression {$$ = $1;}
| expression ',' assignment_expression {}
;

constant_expression
: conditional_expression { printf("constant_expression : conditional_expression\n");
$$ = $1;
}
;
```

```
declaration
: declaration_specifiers ';' {printf("declaration : declaration_specifiers ';'\n");}
| declaration_specifiers init_declarator_list ';' { printf("declaration : declaration_specifiers init_declarator_list ';'\n");
std::list<VAR_DATA *>::iterator it;
int typ = get_type($1);
/*cout << "Declaring variables : " << endl;
cout << "Type : " << typ << "\n";*/
for(it = $2->begin(); it != $2->end(); it++){
symrec *s;
/*cout << *(*it)->name << endl;*/
if(isarray((*it)->typ))
installarray((*it)->name,typ | VAR_SPEC | ((*it)->typ & ~0x1F), (*it)->val.intval);
else
if(isfunction((*it)->typ))
s = install((*it)->name, typ | FUNC_SPEC | ((*it)->typ & ~0x1F));
else{
s = install((*it)->name, typ | VAR_SPEC | ((*it)->typ & ~0x1F));
if((*it)->init)
gencode(STORE_IR, (*it)->var, nullptr, s);
}
}
}
;


declaration_specifiers
: storage_class_specifier { std::list<int> *newlist = new std::list<int>;
newlist->push_back($1);
$$ = newlist;
}
| storage_class_specifier declaration_specifiers { $$ = $2;
```

```
$$->push_back($1);
}
| type_specifier { std::list<int> *newlist = new std::list<int>;
newlist->push_back($1);
$$ = newlist;
}
| type_specifier declaration_specifiers { $$ = $2;
$$->push_back($1);
}
| type_qualifier { std::list<int> *newlist = new std::list<int>;
newlist->push_back($1);
$$ = newlist;
}
| type_qualifier declaration_specifiers { $$ = $2;
$$->push_back($1);
}
;


init_declarator_list
: init_declarator { std::list<VAR_DATA *> *newlist = new std::list<VAR_DATA *>;
VAR_DATA *newstruc = $1;
newlist->push_back(newstruc);
$$ = newlist;
}
| init_declarator_list ',' init_declarator { VAR_DATA *newstruc = $3;
$$ = $1;
$$->push_back(newstruc);
}
;
```

```
init_declarator
: declarator { $$ = $1;}
| declarator '=' initializer { $$ = $1;
$$->init = 1;
$$->var = $3.var;
}
;

storage_class_specifier
: TYPEDEF {
$$ = TYPENAME_SPEC;
}
| EXTERN {
$$ = EXTERN_SPEC;
}
| STATIC {
$$ = STATIC_SPEC;
}
| AUTO {
$$ = AUTO_SPEC;
}
| REGISTER {
$$ = REGISTER_SPEC;
}
;

type_specifier
: VOID {$$ = VOID_SPEC;}
| CHAR {$$ = CHAR_SPEC;}
| SHORT {$$ = SHORT_SPEC;}
```

```
| INT {$$ = INT_SPEC;}
| LONG {$$ = LONG_SPEC;}
| FLOAT {$$ = FLOAT_SPEC;}
| DOUBLE {$$ = DOUBLE_SPEC;}
| SIGNED {$$ = SIGNED_SPEC;}
| UNSIGNED {$$ = UNSIGNED_SPEC;}
| struct_or_union_specifier {}
| enum_specifier {$$ = ENUM_SPEC;}
| TYPE_NAME {$$ = TYPENAME_SPEC;}
;

struct_or_union_specifier
: struct_or_union IDENTIFIER '{' struct_declaration_list '}'
| struct_or_union '{' struct_declaration_list '}'
| struct_or_union IDENTIFIER
;

struct_or_union
: STRUCT
| UNION
;

struct_declaration_list
: struct_declaration
| struct_declaration_list struct_declaration
;

struct_declaration
: specifier_qualifier_list struct_declarator_list ';'
```

```
;

specifier_qualifier_list
: type_specifier specifier_qualifier_list {
$$ = $2;
$$->push_back($1);
}
| type_specifier {
std::list<int> *newlist = new std::list<int>;
newlist->push_back($1);
$$ = newlist;
}
| type_qualifier specifier_qualifier_list {
$$ = $2;
$$->push_back($1);
}
| type_qualifier {
std::list<int> *newlist = new std::list<int>;
newlist->push_back($1);
$$ = newlist;
}
;

struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
;

struct_declarator
```

```
: declarator
| ':' constant_expression
| declarator ':' constant_expression
;

enum_specifier
: ENUM '{' enumerator_list '}'
| ENUM IDENTIFIER '{' enumerator_list '}'
| ENUM IDENTIFIER
;

enumerator_list
: enumerator
| enumerator_list ',' enumerator
;

enumerator
: IDENTIFIER
| IDENTIFIER '=' constant_expression
;

type_qualifier
: CONST {
$$ = CONST_SPEC;
}
| VOLATILE {
$$ = VOLATILE_SPEC;
}
;
```

```
declarator
: pointer direct_declarator {$$ = $2;}
| direct_declarator {$$ = $1;}
;

direct_declarator
: IDENTIFIER {
/*printf("IDENTIFIER = %s\n", $1);*/
printf("direct_declarator : IDENTIFIER\n");
VAR_DATA *newsym = new VAR_DATA();
newsym->name = new string($1);
$$ = newsym;
}
| '(' declarator ')' {printf("direct_declarator : '(' declarator ')'\n");}
| direct_declarator '[' constant_expression ']' {printf("direct_declarator : direct_declarator '['
constant_expression ']'\n");}
| direct_declarator '[' ']' {printf("direct_declarator : direct_declarator '(' parameter_type_list ')'\n");}
| direct_declarator '(' parameter_type_list ')' { printf("direct_declarator : direct_declarator '('
parameter_type_list ')'\n");
$$ = $1;
$$->typ |= FUNC_SPEC;
$$->arglist = $3;
$$->init = 0;
}
| direct_declarator '(' identifier_list ')' {printf("direct_declarator : direct_declarator '(' identifier_list
')'\n");}
| direct_declarator '(' ')' {
printf("direct_declarator : direct_declarator '(' ')'\n");
$$ = $1;
$$->typ |= FUNC_SPEC;
```

```
}
;


pointer
: '*'
| '*' type_qualifier_list
| '*' pointer
| '*' type_qualifier_list pointer
;


type_qualifier_list
: type_qualifier
| type_qualifier_list type_qualifier
;



parameter_type_list
: parameter_list {$$ = $1;}
| parameter_list ',' ELLIPSIS {}
;


parameter_list
: parameter_declaration {
std::list<VAR_DATA *> *newlist = new std::list<VAR_DATA *>;
VAR_DATA *newstruc = new VAR_DATA();
*newstruc = $1;
newlist->push_back(newstruc);
$$ = newlist;
}
```

```
| parameter_list ',' parameter_declaration {
$$ = $1;
VAR_DATA *newstruc = new VAR_DATA();
*newstruc = $3;
$$->push_back(newstruc);
}
;

parameter_declaration
: declaration_specifiers declarator {
$$.name = $2->name;
if(!isarray($2->typ))
$$.typ = get_type($1);
else
$$.typ = $$.typ = get_type($1) | ARRAY_SPEC;
}
| declaration_specifiers abstract_declarator {}
| declaration_specifiers {}
;

identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;

type_name
: specifier_qualifier_list { printf("type_name : specifier_qualifier_list\n");
$$ = get_type($1);
}
```

```
| specifier_qualifier_list abstract_declarator {printf("type_name : specifier_qualifier_list
abstract_declarator\n");}
;

abstract_declarator
: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
;

direct_abstract_declarator
: '(' abstract_declarator ')'
| '[' ']'
| '[' constant_expression ']'
| direct_abstract_declarator '[' ']'
| direct_abstract_declarator '[' constant_expression ']'
| '(' ')'
| '(' parameter_type_list ')'
| direct_abstract_declarator '(' ')'
| direct_abstract_declarator '(' parameter_type_list ')'
;

initializer
: assignment_expression {$$ = $1;}
| '{' initializer_list '}' {}
| '{' initializer_list ',' '}' {}
;

initializer_list
```

```
: initializer
| initializer_list ',' initializer
;


P : /**/
;
statement
: labeled_statement {
$$ = $1;
$$.nextlist = NULL;
}
| P {
enter_scope();
}
compound_statement {
printf("statement : compound_statement\n");
printlocalvars();
exit_scope();
$$ = $3;
}
| expression_statement {
$$.nextlist = NULL;
$$.breaklist = NULL;
$$.caselist = NULL;
}
| selection_statement {
$$ = $1;
}
| iteration_statement {
$$ = $1;
}
```

```
| jump_statement {
$$ = $1;
}
;

labeled_statement
: IDENTIFIER ':' statement {}
| CASE constant_expression ':' N statement {
$$.breaklist = $5.breaklist;
if(!isconstant($2.typ))
yyerror("Error : CASE expression must be constant\n");
if(!isintegral($2.typ))
yyerror("Error : CASE expression must be integral constant\n");
$$.caselist = makecaselist($2.val, $2.typ, $4);
}
| DEFAULT ':' N statement {
$$.breaklist = $4.breaklist;
$$.caselist = makecaselist($3);
}
;

compound_statement
: '{' '}' { printf("compound_statement : '{' '}'\n");
$$.breaklist = NULL;
$$.nextlist = NULL;
}
| '{' statement_list '}' { printf("compound_statement : '{' statement_list '}'\n");
$$ = $2;
}
| '{' declaration_list '}' { printf("compound_statement : '{' declaration_list '}'\n");
```

```
$$.breaklist = NULL;
$$.nextlist = NULL;
}
| '{' declaration_list statement_list '}' { printf("compound_statement : '{' declaration_list statement_list '}'\n");
$$ = $3;
}
;

declaration_list
: declaration
| declaration_list declaration
;

statement_list
: statement {$$ = $1;}
| statement_list N statement {
backpatch($1.nextlist, $2);
$$ = $1;
$$.nextlist = $3.nextlist;
$$.breaklist = merge($1.breaklist, $3.breaklist);
$$.caselist = merge($1.caselist, $3.caselist);
}
;

expression_statement
: ';' { printf("expression_statement : ';'\n");
$$.truelist = nullptr;
$$.falselist = nullptr;
}
```

```
| expression ';' { printf("expression_statement : expression ';'\n");
$$ = $1;
}
;

logic_expression : expression {
$$ = $1;
/*Verificar si expression es comparacion u operacion logica*/
if($1.truelist == nullptr || $1.falselist == nullptr){
std::string *name = newtemp(); /*Nueva variable temporal*/
symrec *s0;
s0 = putsym(name, INTVAL | CONST_SPEC);
*(int *)(datablock+s0->offset) = 0;
$$.truelist = makelist(nextinstr);
gencode(IF_NE_IR, $1.var, s0, nullptr);
$$.falselist = makelist(nextinstr);
gencode(GOTO_IR);
}
}

M : /*EMpty*/ { /*Retornar direccion de siguiente instruccion de IR y dejar hueco para poner la instruccion de salto*/
$$.nextlist = makelist(nextinstr);
gencode(GOTO_IR);
}
;

selection_statement
: IF '(' logic_expression ')' N statement %prec NO_ELSE {
backpatch($3.truelist, $5);
```

```
$$.nextlist = merge($3.falselist, $6.nextlist);
$$.breaklist = $6.breaklist;
}
| IF '(' logic_expression ')' N statement ELSE M N statement {
backpatch($3.truelist, $5);
backpatch($3.falselist, $9);
$$.nextlist = merge($6.nextlist, $8.nextlist, $10.nextlist);
$$.breaklist = merge($6.breaklist, $10.breaklist);
}
| SWITCH M '(' expression ')' statement {
if(isintegral($4.typ))
yyerror("Error : switch expression must be integral\n");
$6.breaklist = merge($6.breaklist, makelist(nextinstr));
gencode(GOTO_IR);
backpatch($2.nextlist, nextinstr);
std::list<CASE_DATA *>::iterator it;
if($6.caselist != NULL){
for(it = $6.caselist->begin(); it != $6.caselist->end(); it++){
int addr = (*it)->addr;
int typ = (*it)->typ;
if(typ == VOIDVAL)
gencode(GOTO_IR, addr);
else{
std::string *name = newtemp();
symrec *s = putsym(name, INTVAL | CONST_SPEC);
symrec *s1 = $4.var;
*(int *)((char *)datablock+s->offset) = (*it)->val.intval;
if(isconstant($4.typ)){
name = newtemp();
s1 = putsym(name, INTVAL | CONST_SPEC);
*(int *)((char *)datablock+s1->offset) = $4.val.intval;
```

```
}
gencode(IF_EQ_IR, s, s1, addr);
}
}
}
backpatch($6.breaklist, nextinstr);
$$.breaklist = NULL;
$$.nextlist = NULL;
}
;

iteration_statement
: WHILE N '(' logic_expression ')' N statement { printf("iteration_statement : WHILE '(' expression ')' statement\n");
backpatch($7.nextlist, $2);
backpatch($4.truelist, $6);
$$.nextlist = $4.falselist;
gencode(GOTO_IR, $2);
backpatch($7.breaklist, nextinstr);
$$.breaklist = NULL;
}
| DO N statement WHILE N '(' logic_expression ')' ';' { printf("iteration_statement : DO statement WHILE '(' expression ')' ';'\n");
backpatch($7.truelist, $2);
backpatch($3.nextlist, $5);
backpatch($3.breaklist, nextinstr);
$$.breaklist = NULL;
$$.nextlist = $7.falselist;
}
| FOR '(' expression_statement N expression_statement N ')' statement { printf("iteration_statement : FOR '(' expression_statement expression_statement ')' statement\n");
backpatch($5.truelist, $6);
```

backpatch($8.nextlist, $4);

$$.nextlist = $5.falselist;

gencode(GOTO_IR, $4);

backpatch($8.breaklist, nextinstr);

$$.breaklist = NULL;

}

| FOR '(' expression_statement N expression_statement N expression ')'

M N statement { printf("iteration_statement : FOR '(' expression_statement expression_statement expression ')' statement\n");

backpatch($5.truelist, $10); /*Completar saltos en verdadero de comparacion a N3*/

backpatch($11.nextlist, $6); /*Completar saltos de statement a N2*/

backpatch($9.nextlist, $4); /*Completar saltos de Ma N1*/

$$.nextlist = $5.falselist; /*Propagar saltos en falso de comparacion*/

gencode(GOTO_IR, $6); /*Insertar salto a N2*/

backpatch($11.breaklist, nextinstr);

$$.breaklist = NULL;

}

;

jump_statement

: GOTO IDENTIFIER ';' {}

| CONTINUE ';' {}

| BREAK ';' {

$$.breaklist = makelist(nextinstr);

$$.nextlist = NULL;

$$.caselist = NULL;

gencode(GOTO_IR);

}

| RETURN ';' {

gencode(RET_IR);

$$.breaklist = NULL;

```
$$.nextlist = NULL;
$$.caselist = NULL;
}
| RETURN expression ';' {
gencode(RET_IR, $2.var, nullptr, nullptr);
$$.breaklist = NULL;
$$.nextlist = NULL;
$$.caselist = NULL;
}
;

translation_unit
: external_declaration {printf("translation_unit--external_declaration \n");}
| translation_unit external_declaration {printf("translation_unit--translation_unit
external_declaration \n");}
;

external_declaration
: function_definition {printf("external declaration--function_definition\n");}
| declaration {printf("external declaration--declaration\n");}
;

function_definition
: declaration_specifiers declarator declaration_list compound_statement
{printf("function_definition : declaration_specifiers declarator declaration_list
compound_statement\n");}
| declaration_specifiers declarator {
int typ = get_type($1) | $2->typ;
curr_func = install($2->name, typ);
gencode(PROC_IR, curr_func);
```

```
localsyms = new symboltable();
enter_scope();
installarg($2->arglist);
/*Iniciar nueva tabla de etiquetas*/
lab_table.clear();
setargs();
}
compound_statement {
printf("function_definition : declaration_specifiers declarator compound_statement\n");
printlocalvars();
exit_scope();
curr_func->func_desc.sym_table = localsyms;
gencode(ENDPROC_IR, curr_func);
}
| declarator declaration_list compound_statement {printf("function_definition : declarator declaration_list compound_statement\n");}
| declarator {
int typ = $1->typ;
curr_func = install($1->name, typ);
gencode(PROC_IR, curr_func);
localsyms = new symboltable();
enter_scope();
/*Iniciar nueva tabla de etiquetas*/
lab_table.clear();
setargs();
}
compound_statement {
printf("function_definition : declarator compound_statement\n");
printlocalvars();
exit_scope();
curr_func->func_desc.sym_table = localsyms;
gencode(ENDPROC_IR, curr_func);
```

```
}
;


%%


void yyerror(const char *message){
printf("\nError: %s at line %d clumn %d\n",message, yylloc.first_line+1, yylloc.first_column+1);
exit(1);
}


int main(int argc, char *argv[]){


string fname; /*Para guardar nombre de archivo compilandose*/
/*Crear tabla de simbolos globales*/
sym_table = new symboltable();
cout<<"Integrantes del equipo:\nSamuel Jesus Ramos Andrade\nGiovanni Daniel Aguirre Salinas\nAlejandro Hernandez Baca\n\n";
cout<<"Compilador de C version 1.0\n";
if(argc > 1){
yyin = fopen(argv[1],"r");
fname = argv[1];
}
else{
cout<<"Uso: "<< argv[0] <<" <filename>\n";
return (0);
}
yyparse();
printvars();
```

```
print_icode();
print_code(fname);
return 0;
}
```

Pruebas:

Es importante mencionar que, para la correcta realización de pruebas, los programas usados no deben contener las librerías, es por ello que en algunas ocasiones las comentamos y en otras simplemente no las pusimos.

Código de Prueba 1 realizada por Samuel Jesús Ramos Andrade:

La prueba más simple y sin embargo de suma importancia en el mundo de los programadores... ¿Quién no ha empezado un lenguaje con su respectivo hola mundo? Pues que sea un hola mundo la primera prueba para nuestro compilador.

```
/*#include <stdio.h>*/
void printf();
int main(int argc, char *argv[]){
printf("Hola Mundo");
}
```

Resultando en la generacion correcta de nuestros arboles

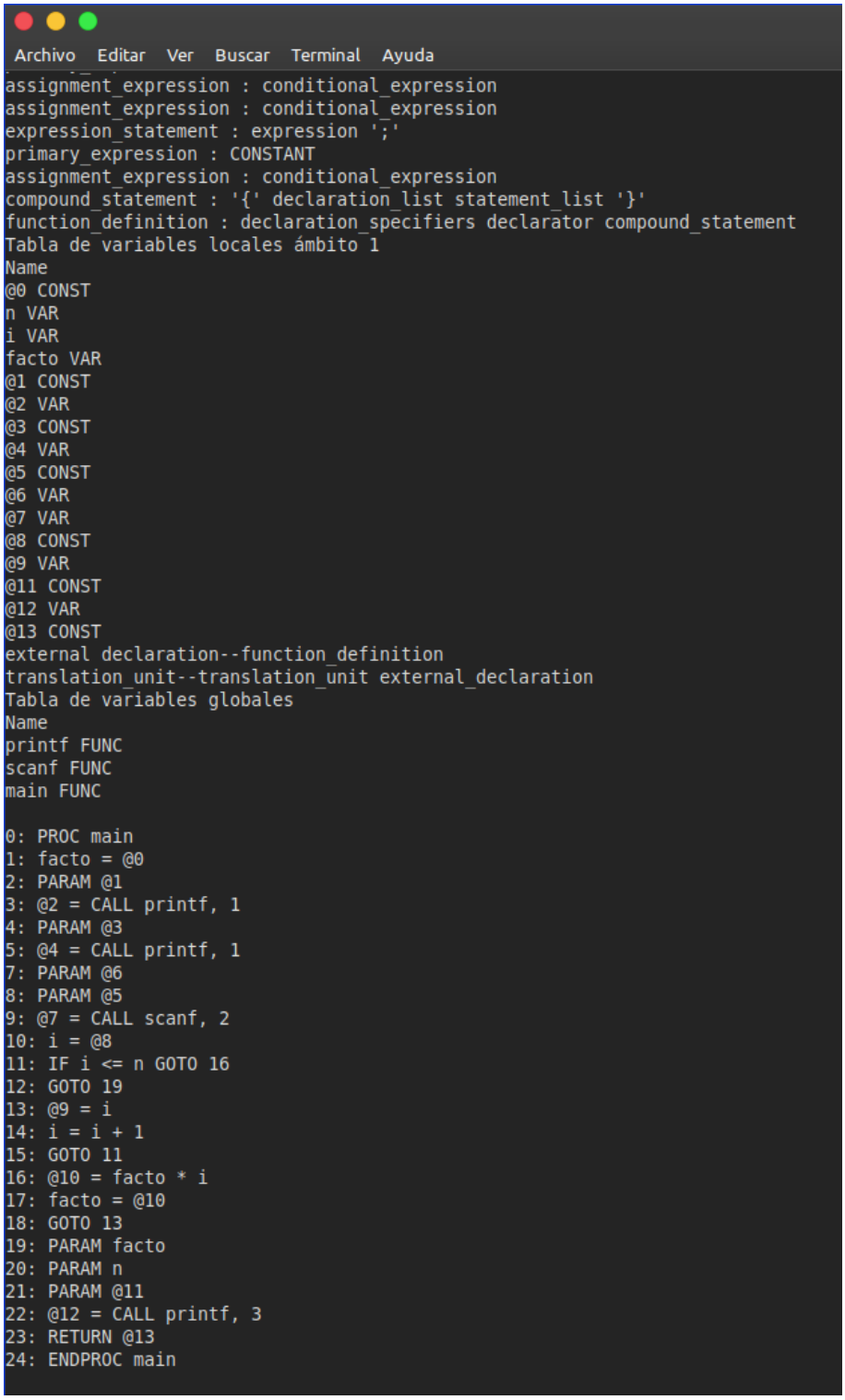
Prueba 2 realizada por Alejandro Hernández Baca.

El siguiente programa tiene como fin calcular la factorial de un numero dado, en la captura podemos observar la generación de tabla de variables, así como el código intermedio del mismo.

factorial.c

```
/*

Programa  (Leer un numero n y calcular la multiplicacion de 1 a n con un for)

#include<stdio.h>*/

void printf();

void scanf();

int main()

{

int n,i,facto=1;

printf("Programa que hace el factorial de un numero n\n\n");

printf("Ingrese un numero n:\n");

scanf("%i",&n);
```

```
for(i=1;i<=n;i++)
{
facto*=i;
}
printf("El factorial de %i es: %i\n\n",n,facto);

return 0;
}
```



Prueba 3 realizada por Alejandro Hernández Baca.

Código.c

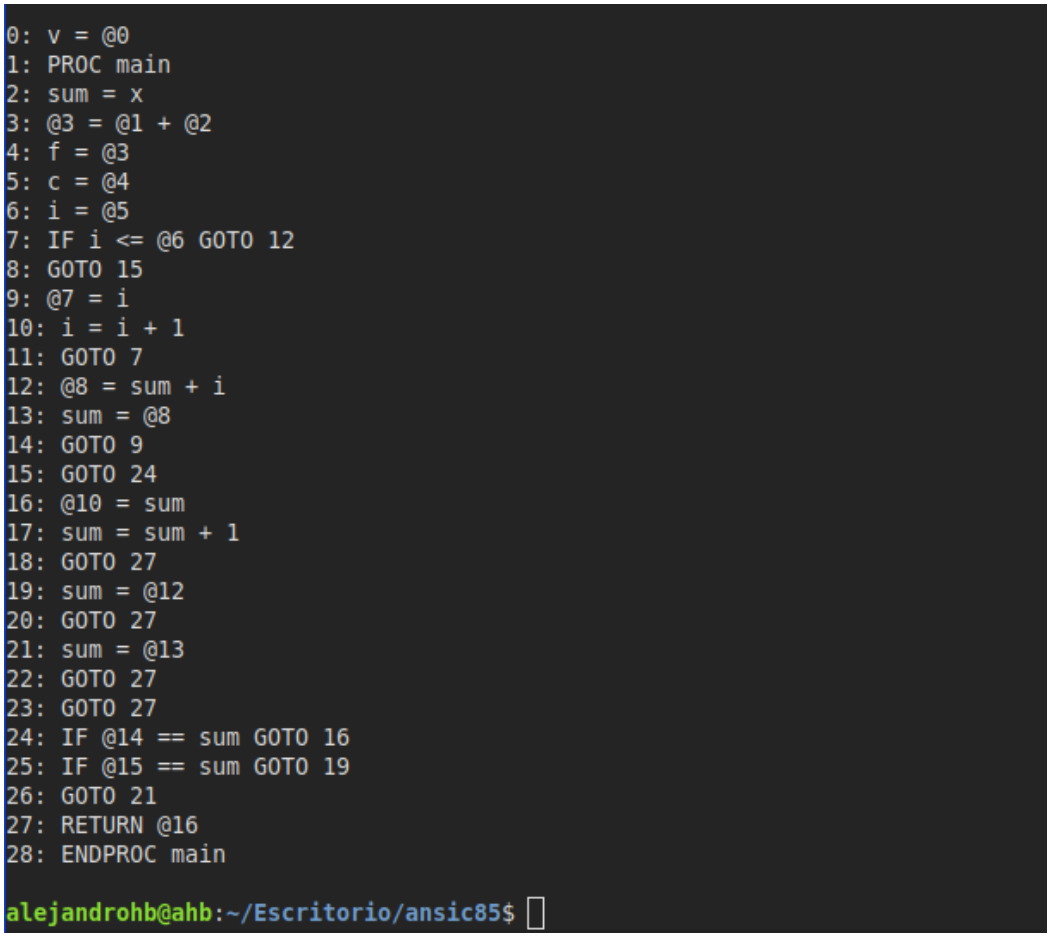
En este pequeño programa se realizan varias pruebas para el compilador, como lo es un for y un switch.

```
/* add.c
* a simple C program
*/

int x, y, z, u, v = 0;

void printf();
int main(int argc, char *argv[])
{
int i, sum = x;
double f = 1.5 + 2;
char* c = "\x41";
for( i = 1; i <= 20; i++ ) {
int j;
sum += i;
} /*-for-*/
switch(sum){
case 0:
sum++;
break;
case 1:
sum = 0;
break;
default:
sum = sizeof(int);
break;
}
return 0;
}
```

Podemos observar que las tablas de variables son generadas de manera correcta, así como las variables mismas, usadas a lo largo del programa. En cuanto al código

intermedio, podemos observar cómo es generado de tal manera que puede ser entendido si lo leemos instrucción por instrucción.

```
Tabla de variables locales ámbito 2
Name
@9 CONST
@10 VAR
@11 CONST
@12 CONST
@13 CONST
primary_expression : CONSTANT
assignment_expression : conditional_expression
compound_statement : '{' declaration_list statement_list '}'
function_definition : declaration_specifiers declarator compound_statement
Tabla de variables locales ámbito 1
Name
argc VAR
argv VAR
i VAR
sum VAR
@1 CONST
@2 CONST
@3 VAR
f VAR
@4 CONST
c VAR
@5 CONST
@6 CONST
@7 VAR
@14 CONST
@15 CONST
@16 CONST
external declaration--function_definition
translation_unit--translation_unit external_declaration
Tabla de variables globales
Name
@0 CONST
x VAR
y VAR
z VAR
u VAR
v VAR
printf FUNC
main FUNC
```



Prueba 4 realizada por: Giovanni Daniel Aguirre Salinas

Para esta prueba se utiliza un código el cual contiene un ciclo do-while, el cual recibe un número y si no es el que se está pidiendo incrementa la cantidad de la variable suma.

Codigo:

```
void printf();

Void scanf();

int main()

{

    int x;

    int suma=0;

    do

    {

        printf("Ingrese un numero(si desea salir ingrese 0):\n");

        scanf("%d",&x);

        suma+=x;
```

```
    } while ( x != 0);
    printf("\nLa suma es: %d\n", suma);
    return 0;
}
```

Resultados de la prueba:

Como se puede observar las variables son detectadas correctamente y el código intermedio es generado correctamente.

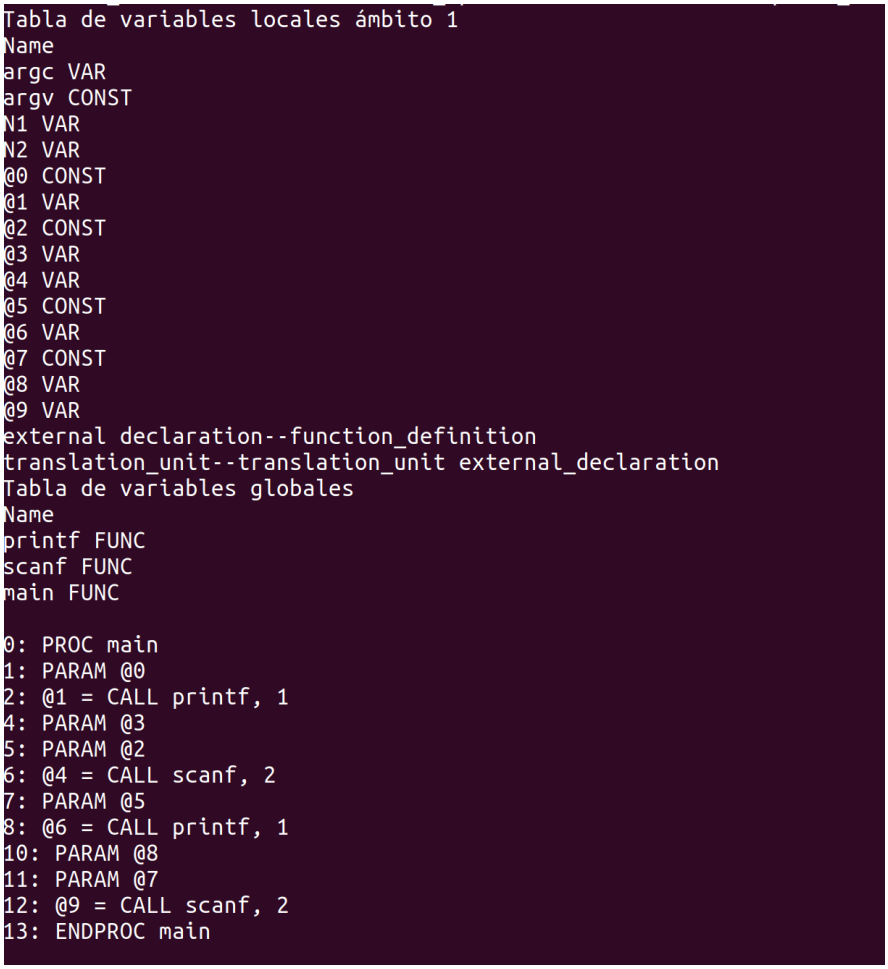
```
Tabla de variables locales ámbito 2
Name
@1 CONST
@2 VAR
@3 CONST
@4 VAR
@5 VAR
@6 VAR
primary_expression : IDENTIFIER
primary_expression : CONSTANT
assignment_expression : conditional_expression
iteration_statement : DO statement WHILE '(' expression ')' ';'
primary_expression : IDENTIFIER
primary_expression : STRING_LITERAL
assignment_expression : conditional_expression
primary_expression : IDENTIFIER
assignment_expression : conditional_expression
assignment_expression : conditional_expression
expression_statement : expression ';'
primary_expression : CONSTANT
assignment_expression : conditional_expression
compound_statement : '{' declaration_list statement_list '}'
function_definition : declaration_specifiers declarator compound_statement
```

Prueba 5 realizada por: Samuel Jesus Ramos Andrade:

Esta prueba es simple, solo es la asignación de dos números, con printf y scanf como funciones.

Código:

```
void printf();

void scanf();

int main(int argc, char const *argv[])

{

int N1,N2;

printf("Dame el primer numero");

scanf("%i",&N1);

printf("Dame el segundo numero");

scanf("%i",&N2);

}
```



Prueba 6 realizada por Samuel Jesus Ramos Andrade:

Esta prueba ya es un poco más compleja, es para sacar el cuadrado de un numero entero mediante la función pow(), además de probar si reconoce las variables de tipo flotante, aunque no se usen.

Código:

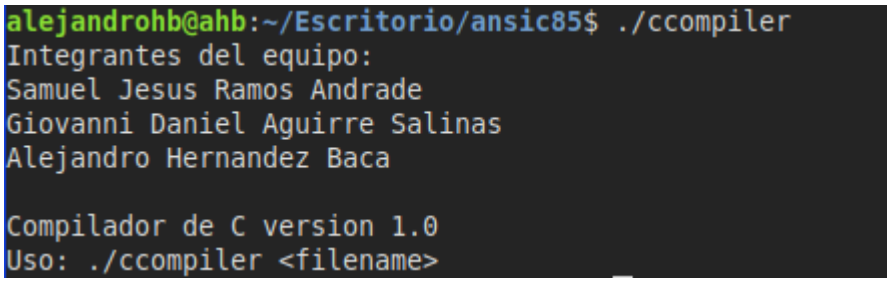
```
void printf();

void scanf();

void pow();

int main(int argc, char const *argv[])

{

int N1,N2, multip, cuad1,cuad2;

float raiz1,raiz2;

printf("Dame el primer numero");
```

```
scanf("%i",&N1);
printf("Dame el segundo numero");
scanf("%i",&N2);
multip = N1*N2;
cuad1 = pow(N1);
cuad2 = pow(N2);
printf("%i",cuad1);
printf("%i",cuad2);
}
```

```
Tabla de variables locales ámbito 1
Name
argc VAR
argv CONST
N1 VAR
N2 VAR
multip VAR
cuad1 VAR
cuad2 VAR
raiz1 VAR
raiz2 VAR
@0 CONST
@1 VAR
@2 CONST
@3 VAR
@4 VAR
@5 CONST
@6 VAR
@7 CONST
@8 VAR
@9 VAR
@10 VAR
@11 VAR
@12 VAR
@13 CONST
@14 VAR
@15 CONST
@16 VAR
external declaration--function_definition
translation_unit--translation_unit external_declaration
Tabla de variables globales
Name
printf FUNC
scanf FUNC
pow FUNC
main FUNC

0: PROC main
1: PARAM @0
2: @1 = CALL printf, 1
4: PARAM @3
5: PARAM @2
6: @4 = CALL scanf, 2
7: PARAM @5
8: @6 = CALL printf, 1
10: PARAM @8
11: PARAM @7
12: @9 = CALL scanf, 2
13: @10 = N1 * N2
14: multip = @10
15: PARAM N1
16: @11 = CALL pow, 1
17: cuad1 = @11
18: PARAM N2
19: @12 = CALL pow, 1
20: cuad2 = @12
21: PARAM cuad1
22: PARAM @13
23: @14 = CALL printf, 2
24: PARAM cuad2
25: PARAM @15
26: @16 = CALL printf, 2
27: ENDPROC main
```

Prueba 7 realizada por Alejandro Hernández Baca.

En esta prueba se ejecuta nuestro compilador, pero sin enviarle ningún archivo a compilar, de esta manera, probamos que nos lanza un mensaje con la manera correcta de usar nuestro compilador.



Prueba 8 realizada por: Giovanni Daniel Aguirre Salinas.

Para la octava y última prueba usaremos un programa sencillo el cual pide 3 números al usuario y decide cual él es mayor de los 3 y lo muestra en la pantalla.

```
void printf();

void scanf();


int main() {

double n1, n2, n3;

printf("Ingrese tres numeros diferentes: ");

scanf("%lf %lf %lf", &n1, &n2, &n3);

/* si n1 es el mas grande */

if (n1 >= n2 && n1 >= n3)

printf("%.2f es el numero mas grande.", n1);


/* si n2 es el mas grande */

if (n2 >= n1 && n2 >= n3)

printf("%.2f es el numero mas grande.", n2);

/* si n3 es el mas grande */

if (n3 >= n1 && n3 >= n2)
```

printf("%.2f es el numero mas grande.", n3);

return 0;

}

```
Tabla de variables locales ámbito 1
Name
n1 VAR
n2 VAR
n3 VAR
@0 CONST
@1 VAR
@2 CONST
@3 VAR
@4 VAR
@5 VAR
@6 VAR
@7 CONST
@8 VAR
@9 CONST
@10 VAR
@11 CONST
@12 VAR
@13 CONST
external declaration--function_definition
translation_unit--translation_unit external_declaration
```

```
Tabla de variables globales
Name
printf FUNC
scanf FUNC
main FUNC

0: PROC main
1: PARAM @0
2: @1 = CALL printf, 1
6: PARAM @5
7: PARAM @4
8: PARAM @3
9: PARAM @2
10: @6 = CALL scanf, 4
11: IF n1 < n2 GOTO 13
12: GOTO 18
13: IF n1 < n3 GOTO 15
14: GOTO 18
15: PARAM n1
16: PARAM @7
17: @8 = CALL printf, 2
18: IF n2 < n1 GOTO 20
19: GOTO 25
20: IF n2 < n3 GOTO 22
21: GOTO 25
22: PARAM n2
23: PARAM @9
24: @10 = CALL printf, 2
25: IF n3 < n1 GOTO 27
26: GOTO 32
27: IF n3 < n2 GOTO 29
28: GOTO 32
29: PARAM n3
30: PARAM @11
31: @12 = CALL printf, 2
32: RETURN @13
33: ENDPROC main
```

Conclusiones

Giovanni Daniel Aguirre Salinas:

Para finalizar podemos decir que el desarrollo de un compilador es bastante complicado, por lo que se debe tener especial cuidado con cada una de las partes que lo confirman, ya que si alguna tiene un fallo todo saldrá mal, es por eso que este proyecto es de gran ayuda para poder formarnos como buenos programadores.

Samuel Jesús Ramos Andrade:

El desarrollo de un compilador es una tarea compleja, desde el analizador léxico hasta el analizador sintáctico, las pruebas y el aprendizaje que nos deja esta práctica es de valor para la carrera, ya que al programar entendíamos lo que hacia nuestro programa y no lo que hacía el compilador al recibir las instrucciones de nuestro programa. La parte sobre todo del analizador léxico fue donde se complicó un poco más por ser una base potencial para la identificación de los tokens para el analizador sintáctico.

Alejandro Hernández Baca:

A lo largo de la realización de nuestro compilador, tuvimos bastante en cuenta todo lo visto en el curso, con la aplicación de ello, fue que pudimos reforzar el aprendizaje, al mismo tiempo que comprendíamos todo el proceso que hay detrás de correr un simple programa en C, que hasta el día de hoy era una tarea sencilla para nosotros, pero todo ese proceso es realmente complejo a la vez que increíble.