Compiler Design Project
Part Two

Semantic Analysis

Project Guided by :

Prof P. Santhi Thilagam

Project done by :

Menezes Elvis Edward (15CO254)
Somnath Krishna Sarkar (15CO247)

Contents.....

Compiler Design

The Compiler Design Process is broken down mainly into two parts namely:

1. Front-End
    a) Lexical Analysis
    b) Syntax Analysis
    c) Semantic Analysis

2. Back-End
    a)Code Optimizer
    b)Code Generator

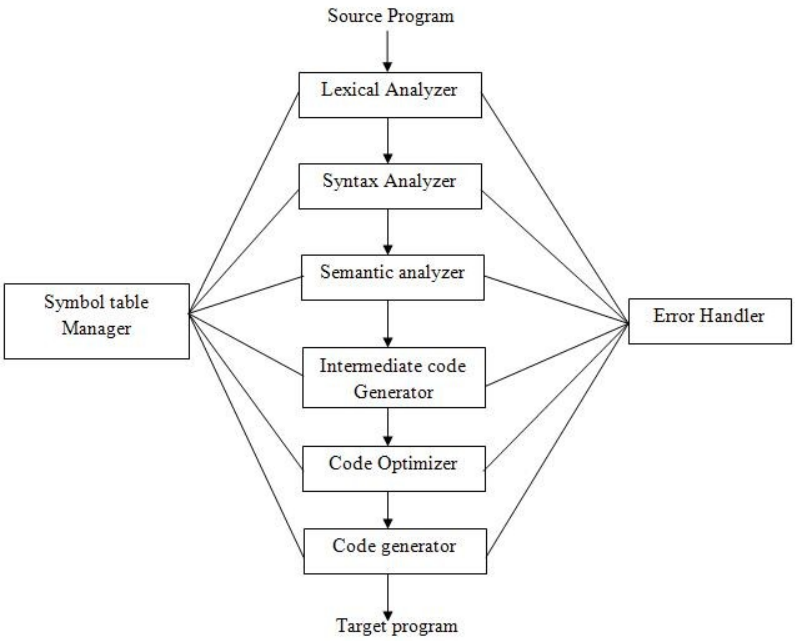


Fig 1.3 Phases of a Compiler

Lexical Analysis:

Lexical Analysis is the first phase of compiler. It is also known as scanner. The input program is converted into a sequence of tokens that can be treated as a unit in the grammar of the programming languages.

Syntax Analysis:

The second phase of the compiler is Syntax Analysis also known as Parsing. The syntactical structure of the given input is checked in this phase, i.e. whether it is the correct syntax. A data structure is built and used in this process called as a Parse tree and also as Syntax tree. The parse tree is constructed by using the predefined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax.

Semantic Analysis:

In this phase the parse tree is verified, if it is meaningful or not and produces a verified parse tree.

Intermediate Code Generator:

A form which can be readily executed by machine is generated in this phase of the compiler. One of the many popular intermediate codes is the Three address code.

Code Optimizer:

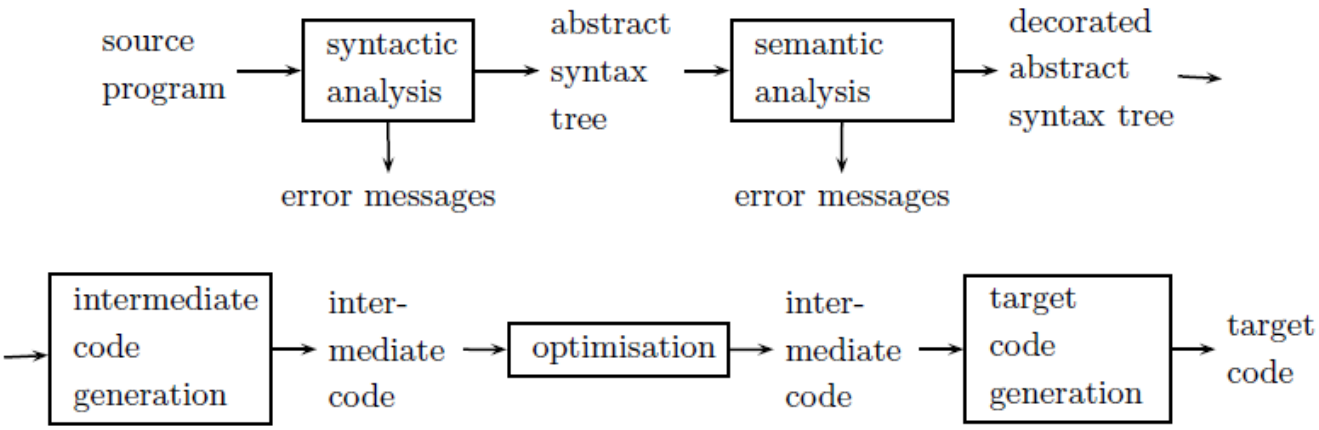
In this phase the code is transformed into more efficient code so that it consumes less resources and is executed faster without the meaning of the code not altered. Optimization can be categorized into two types: machine dependent and machine independent.

Code Generator:

This is the final stage of the compiler and the main purpose of this stage is write a code that the machine can understand. The output is dependent on the assembler.

This Report mainly focuses on the second phase of the compiler that is the semantic analyzer.

Semantic Analyzer:



Semantic Analyser:

The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.

Hence in the semantic analysis phase, meaning is provided to the CFG productions which helps interpret symbols, their types, and their relations with each other.

Hence Semantic analysis checks if the sytax structure of the source program makes any meaning or not. Some of the tasks to be performed in semantic analysis are:

1. Scope Resolution
2. Type Checking
3. Array – bound Checking

Some terms used in semantic phase:

Attribute Grammar - attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.

Types :

1. L – attributed
2. S - attributed

Semantic Errors:

A semantic error is a violation of the rules of meaning of a natural language or a programming language.

Semantic errors usually are detected by the programmer or user of the program, often while reading output and finding it is incorrect.

Most frequent semantic errors are:

1. Function parameters
    1. Parameters of type void are given as errors
    2. Number of parameters and argument list dont match.
    3. Argument type doesn’t match with the parameter type.
2. Identifier cannot be called as function -

    If an identifier which is not declared is called as a function it is a semantic error.
3. No functions defined is also to be handled in the semantic phase.
4. Return -

    Int function returning void or void function returning int are semantic errors and are handled in the semantic phase.
5. Type mismatch

    Bitwise operators can be applied only on int datatype and if a float is used instead it is a semantic error. The semantic analysis phase takes care of this error.
6. Undeclared variable -

    Variables that are addressed but not have been declared before use are undeclared variables and are given as a warning by the semantic phase.
7. Redeclared variable -

    Variables that are declared once and are redeclared are to be outputted as warnings in semantic analysis phase.
8. While test case should be of type int

    The condition inside while loop should be of type int any other type is a semantic error and is handled by the semantic phase.
9. Main not last function
10. Pointer errors: invalid array indices, attempting to dereference a variable which is not a pointer, etc.

Explanation:

Lexical Analyzer

- Handles Single Line comments, Multi-Line comments, Preprocessor Statements, and white space are recognized but not passed to the parser.
- Lexical Errors returned are invalid characters, and dangling string literals.
- Line numbers are calculated here and are used by the hash map and the parser.
- We pass identifier names and datatype keywords as strings to yylval so that the symbol table can be computed on the parser side.
- The constant table is created here, with support for String, Integer and Floating Point constants. The Hash Function used is the summation of ASCII values of characters involved in the constant string (stored in yytext), multiplied by a prime raised to the index of the respective character. The whole hash value is taken modulo a large prime. The Collision handling method used is separate chaining using linked lists.

Syntax + Semantic Analyzer

- The parser was rewritten to eliminate 5 shift/reduce conflicts. Resultant parser has no conflicts and is error-free.
- We redefine yylval as a union of 3 types:

1. String for passing identifier names, datatypes,etc from the lexer.
2. Expression structure, used to evaluate type-compatibilty as well as lvalue assignment errors
3. Symbol Table, used to manage identifiers at a statement and scope level.

- The symbol table has support for line numbers, datatypes, identifier names, function return types, information about number and type of function parameters as well as nesting level and scope path. It also contains information about pointer levels of indirection.
- We support all primitive and compound datatypes found in ANSI C.
- The symbol table implements scope using a stack of symbol tables. When a scope opens, we push the name of the scope (function name, struct/union identifier name, if/else/while block etc.) and a fresh symbol table to the stack. When a scope ends, we apply the scope name to the items in the stack and merge back into the parent symbol table, keeping all information intact. (functions push,pop in parser and the functions in symbol table help with this).
- In order to implement operand checking for all binary and unary operators, we needed to make use of an expression structure: The expression structure has two values, the first is a string representing the datatype of the corresponding operation, the second is a integer representing whether the expression is modifiable or not.
- We use the function check_exp to ensure the validity of an operand against a set of valid possible datatypes. For example bitwise operations are only defined on integer types (char, short, int, long) and not others (float, int pointers, etc.). So at the time of parsing a bitwise expression, we check if the operands against the integer types. If they do not match and error is raised.
- For the pointer operands([], dereference asterisk, arrow) we check if the given expression has at least one level of indirection in the function check_pnt
- For the operands that require a modifiable lvalue, we check mutability using the check_mod function which ensures the modifiable parameter of the expression struct is true.
- In order to update the expression structure we traverse the semantic tree in bottom up fashion, and update the expression depending on the type of operands included. For example, an expression of the type a + 5, would have two expressions a (modifiable: True,

datatype: float) and 5 (modifiable: False, datatype: int). The resultant expression would be defined as a + 5 (modifiable: False, datatype: float)

- We have support for implicit conversion between numeric types. (function iconv in parser.y)
- We implement undeclared identifier error by checking each symbol table in the scope stack until we find the referenced identifier. If it is not found, we raise the error (function lookup in symboltable.c, undeclared_identifier in parser.y)
- We implement redeclared identifier by checking the current scope symbol table alone. If the identifier is already present in the table, error is raised. (function lookup in symboltable.c, redeclared_identifier in parser.y)
- Function parameters are stored as a "function signature" for example int main(int argc, char** argv) is stored as type int(int,char**). This is used for checking function parameters at time they are called. (function dstring in symboltable.c)
- When a identifier is called as a function, we first make sure that the identifier indeed is a function using check_func in parser.y
- Then we format the expressions passed into a function signature, using the expression structure described above. For example a function call of form abc(1,5.0) would have signature (int,float). This would be compared against expected parameters in check_func in parser.y. If there is a type mismatch, or difference in number of expected and received parameters, a semantic error is raised.
- All syntax errors and semantic errors are given a specialized error message with line number received from lexer.

Code for Lexer:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "parser.tab.h"
extern YYSTYPE yylval;
const char * keywords[] =
{"int","float","char","long","short","double","void","if","else","while","struct","for","switch","case","
break","union","return"};
int kc = 17;
int lineno = 1;
const int MOD = 104729,PRIME = 1009;
struct hash_node{
        char *value,*dtype;
        int line;
        struct hash_node *next;
}*hash_map[104729];
int compute_hash(const char *str){
        int ans = 0;
        int len = strlen(str);
        int i;
        for(i = 0; i<len; i++){
                ans*=PRIME;
                ans%=MOD;
                ans+=str[i];
                ans%=MOD;
        }
        return ans;
}
int hash_lookup(const char *str){
        int hash_value = compute_hash(str);
        struct hash_node *focus = hash_map[hash_value];
        while(focus){
                if(!strcmp(focus->value,str))
                        return 1;
                focus = focus->next;
        }
        return 0;
}
char *constant_type(const char *str){
        char *result;
        if(strstr(str,"\""))
                result = strdup("char*");
        else if(strstr(str,"."))
                result = strdup("float");
        else
                result = strdup("int");
        return result;
}
void insert_hash(const char *str){
```

```
        if (hash_lookup(str))
                return;
        int hash_value = compute_hash(str);
        if(hash_map[hash_value]==NULL){
                hash_map[hash_value] = (struct hash_node *)malloc(sizeof(struct hash_node));
                hash_map[hash_value]->value = strdup(str);
                hash_map[hash_value]->dtype = constant_type(str);
                hash_map[hash_value]->line = lineno;
                hash_map[hash_value]->next = NULL;
        }else{
                struct hash_node *focus = hash_map[hash_value];
                while(focus->next)
                        focus = focus->next;
                focus->next = (struct hash_node *)malloc(sizeof(struct hash_node));
                focus->next->value = strdup(str);
                focus->next->dtype = constant_type(str);
                focus->next->line = lineno;
                focus->next->next = NULL;
        }
}
void printconstanttable(){
        printf("Constant Table\n");
    printf("------------------------------------------------------------------------\n");
        printf("|   Constant Value      |   Line              |   Type              |\n");
    printf("------------------------------------------------------------------------\n");
        int i;
        for(i=0; i<MOD; i++){
                struct hash_node *focus = hash_map[i];
                while(focus){
                        printcell(focus->value);
                        char str[30];
                        sprintf(str,"%d",focus->line);
                printcell(str);
                        printcell(focus->dtype);
                        printf("|");
                        printf("\n");
    printf("------------------------------------------------------------------------\n");
                        focus = focus->next;
                }
        }
}
void lexinit(){
        int i;
        for(i = 0; i<MOD; i++)
                hash_map[i] = NULL;
}
%}
letter [a-zA-Z]
digit [0-9]
singlecomment "//".*
multicomment "/*"([^*]|\*[^/])*"*/"
comment {singlecomment}|{multicomment}
danglingcomment "/*"
preprocessor "#".*
ws [\t ]+
```

```
newline [\n]
datatype ("int"|"float"|"double"|"long"|"short"|"char"|"void")
identifier {letter}({digit}|{letter}|[_])*
if_keyword "if"
else_keyword "else"
while_keyword "while"
switch_keyword "switch"|"case"|"break"
for_keyword "for"
return_keyword "return"
keyword {if_keyword}|{else_keyword}|{while_keyword}|{switch_keyword}|{return_keyword}
intliteral {digit}+
floatliteral (({digit}+("."{digit}*)?)|("."{digit}+))
stringliteral "\""(\\.|[^\\"])*["]
literal {intliteral}|{floatliteral}|{stringliteral}
binary_operator "+="|"-="|"*="|"/="|"^="|"&="|"|="|"=="|"&&"|"||"|"->"|"!="|"<="|">="|[+\-
*/\^&|<>]
unary_operator "++"|"--"
semicolon [;]
comma [,]
%%
{singlecomment} ;
{multicomment} {
   int i = 0;
   for(i = 0; i < yyleng; i++)
      lineno+=(yytext[i]=='\n');
}
{preprocessor} ;
{ws} ;
{newline} lineno++;
{identifier} {
   int match = 0,i = 0;
   for(i = 0; i<kc; i++){
      if(strcmp(keywords[i],yytext)==0){
         match = 1;
         break;
      }
   }
   if(match){
      REJECT;
   }else{
      yylval.str = strdup(yytext);
      return ID;
   }
}
"if" return IF;
"else" return ELSE;
"while" return WHILE;
"return" return RETURN;
"struct"|"union" {yylval.str = strdup(yytext); return STRUCTUNION;}
{unary_operator} {
   return UNARY_OPERATOR;
}
"+=" return PE;
"-=" return ME;
"^=" return XE;
```

```
"&=" return AE;
"|=" return OE;
"==" return EQU;
"!=" return NE;
"<=" return LE;
">=" return GE;
"->" return PNT;
"&&" return AA;
"||" return OO;
[+\-*/\^&|<>~] return *yytext;
"=" return EQ;
{datatype} {
   yylval.str = strdup(yytext);
   return DATATYPE;
}
{literal} {
        insert_hash(yytext);
        yylval.exp.mod = 0;
        yylval.exp.dtype = constant_type(yytext);
   return CONSTANT;
}
";" return *yytext;
"(" return BO;
")" return BC;
"{" return *yytext;
"}" return *yytext;
"[" return *yytext;
"]" return *yytext;
{comma} {
   return COMMA;
}
"\r" ;
. {
   printf("Lexical Error: Line %d\n",lineno);
}
%%
```

Code for Parser:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "symboltable.h"
extern FILE* yyin;
extern int lineno;
void lexinit();
void printconstanttable();
int was_return;
int last_func = -1;
char *last_fname;
char datatype[20];
struct symboltable* symstack[100];
const char *allow_types[] = {"char","short","int","long","float","double","void"};
int allow_mask[3][7] = {{1,1,1,1,0,0,0},{1,1,1,1,1,1,0},{0,0,0,0,0,0,1}};
char *scstack[100];
int sp = 0;
void init(){
        lexinit();
   scstack[0] = strdup("");
   symstack[0] = createsymboltable();
}
void push(char *scope){
   symstack[++sp] = createsymboltable();
   scstack[sp] = strdup(scope);
}
void pop(){
        apply_scope(symstack[sp],scstack[sp]);
        merge_table(symstack[sp-1],symstack[sp],0);
   sp--;
}
char * get_scope(){
        char *ret;
        if(sp==0)
                ret = strdup("");
        else{
                ret = strdup(scstack[1]);
                for(int i = 2; i<=sp; i++){
                        char *newret = (char *)malloc(strlen(ret)+1+strlen(scstack[i])+1);
                        strcpy(newret,ret);
                        strcat(newret,".");
                        strcat(newret,scstack[i]);
                        free(ret);
                        ret = newret;
                }
        }
        return ret;
}
void printcell(char *str){
                if(str==NULL)
```

```
                str = strdup("");
            int i = 0;
            int space = 0;
            space = 20 - strlen(str);
        printf("|");
        for(i = 0; i<3; i++)
            printf(" ");
        printf("%s",str);
        for(i =0 ; i<space; i++)
            printf(" ");
}
void printsymboltable(){
        int space = 10;
        int i= 0;
   printf("Symbol Table\n");
   struct node *focus = (symstack[0])->root;

printf("-----------------------------------------------------------------------------------------------\n")
;
   printf("|   Identifier Name     |   Line\t\t|   Datatype\t       |   Scope\t       |\n");

printf("-----------------------------------------------------------------------------------------------\n")
;
   while(focus){
        printcell(focus->id);
        char str[30];
        sprintf(str,"%d",focus->line);
        printcell(str);
        printcell(focus->dtype);
        printcell(focus->scope);
        printf("|");
        printf("\n");


printf("-----------------------------------------------------------------------------------------------\n")
;
      focus = focus->next;
   }
}

char *indirect(char *str){
        char *newstr = (char *)malloc(strlen(str)+2);
        strcpy(newstr,str);
        strcat(newstr,"*");
        return newstr;
}

char *deref(char *str){
        if(!strlen(str)||str[strlen(str)-1]!='*')
            return str;
        char *newstr = (char *)malloc(strlen(str));
        strncpy(newstr,str,strlen(str)-1);
        return newstr;
}
```

```
void check_pnt(char *str){
	if(!strlen(str)||str[strlen(str)-1]!='*')
		printf("Semantic Error: Expression requires pointer type, %s received on Line
%d\n",str,lineno);
}

char *check_func(char *fun, char* params){
	for(int i = sp; i>=0; i--){
		struct node *lu = lookup(symstack[i],fun);
		if(lu!=NULL){
			if(lu->dtype[strlen(lu->dtype)-1]!=')')
				printf("Semantic Error: %s is called as a function, is of type %s on
Line %d\n",lu->id,lu->dtype,lineno);
			else{
				int pnt = 0;
				while(lu->dtype[pnt]!='(')
					pnt++;
				if(strcmp(((lu->dtype)+pnt),params))
					printf("Semantic Error: For function %s got parameters %s,
expected %s on Line %d\n",lu->id,params,(lu->dtype)+pnt,lineno);
				char *ret = (char *)malloc(pnt+1);
				strncpy(ret,lu->dtype,pnt);
				return ret;
			}
		}
	}
	return strdup("void");
}

void check_mod(struct exp_type exp);
struct exp_type undeclared_identifier(char * x);
int check_exp(struct exp_type exp, int am, char *op);
char *iconv(struct exp_type e1, struct exp_type e2);
%}

%code requires {
	struct exp_type{
		int mod;
		char *dtype;
	}exp;
}

%union{
   char * str;
	struct exp_type exp;
	struct symboltable *sym;
}

%token DATATYPE ID CONSTANT IF WHILE RETURN STRUCTUNION
%token BO BC

%token COMMA EQ PE XE AE OE ME OO AA EQU NE LE GE UNARY_OPERATOR PNT LS RS DE

%right "then" ELSE
```

```
%%
START_SYMBOL
	: S
	;
S
	: /* empty */
	| STATEMENT S
	;

STATEMENT
	: EXP ';'
	| FUNCTIONBLOCK
	| DECLARATION ';' {merge_table(symstack[sp],$<sym>1,1);}
	| COMPOUND_STATEMENT
	| IFHEAD STATEMENT %prec "then" {pop();}
	| IFHEAD STATEMENT ELSE {pop(); push("else");} STATEMENT {pop();}
	| WHILEHEAD STATEMENT {pop();}
	| RETURN EXP ';' {was_return = 1;}
	;

COMPOUND_STATEMENT
	: '{' S '}'
	;

IFHEAD
	: IF BO EXP BC {push("if");  check_exp($<exp>3,0,"if test");}
	;

WHILEHEAD
	: WHILE BO EXP BC {push("while"); check_exp($<exp>3,0,"while test");}
	;

DECLARATION
	: DATATYPE DECLIST {assign_datatype($<sym>2,$<str>1); $<sym>$ = $<sym>2;}
	| STRUCTUNION ID '{' DECLARATION_LIST '}' {
		push($<str>2);
		merge_table(symstack[sp],$<sym>4,1);
		pop();
		$<sym>$ = createsymboltable();
		add_identifier($<sym>$,$<str>2,$<str>1);
	}
	| STRUCTUNION ID '{' DECLARATION_LIST '}' DECLIST {
		push($<str>2);
		merge_table(symstack[sp],$<sym>4,1);
		pop();
		char *su_dtype = (char *)malloc(strlen($<str>1)+1+strlen($<str>2)+1);
		strcpy(su_dtype,$<str>1);
		strcat(su_dtype," ");
		strcat(su_dtype,$<str>2);
		assign_datatype($<sym>6,su_dtype);
		free(su_dtype);
		$<sym>$ = createsymboltable();
		add_identifier($<sym>$,$<str>2,$<str>1);
		merge_table($<sym>$,$<sym>6,1);
	}
```

```
	| STRUCTUNION ID DECLIST
	;

DECLARATION_LIST
	: DECLARATION ';' {$<sym>$ = $<sym>1;}
	| DECLARATION DECLARATION_LIST {merge_table($<sym>1,$<sym>2,1); $<sym>$ =
$<sym>1;}
	;

DECLIST
	: DECID EQ EXP
	| DECID
	| DECLIST COMMA DECID {merge_table($<sym>1,$<sym>3,1);}
	| DECLIST COMMA DECID EQ EXP {merge_table($<sym>1,$<sym>3,1);}
	;

DECID
	: ID {$<sym>$ = createsymboltable(); add_identifier($<sym>$,$<str>1,"?");}
	| '*' DECID {apply_indirection($<sym>2);$<sym>$=$<sym>2;}
	;

FUNCTIONBLOCK
	: FUNCTIONHEAD COMPOUND_STATEMENT {
		pop();
		if(!strcmp(datatype,"void")){
			if(was_return)
				printf("Semantic Warning Found return in function of type void on Line
%d\n",lineno);
		}else if(!was_return)
			printf("Semantic Warning No return found in function of type %s on Line
%d\n",datatype,lineno);
	}
	| FUNCTIONHEAD ';' {pop();}
	;

FUNCTIONHEAD
	: DATATYPE DECID BO BC {
		char* functype = (char *)malloc(strlen($<str>1)+3);
		strcpy(functype,$<str>1);
		strcpy(datatype,$<str>1);
		strcat(functype,"()");
		assign_datatype($<sym>2,functype);
		free(functype);
		char* funcname = strdup($<sym>2->root->id);
		last_func = lineno;
		last_fname = strdup(funcname);
		merge_table(symstack[sp],$<sym>2,1);
		push(funcname);
		free(funcname);
		was_return = 0;
	}
	| DATATYPE DECID BO ARGLIST BC {
		char* funcargs = dstring($<sym>4);
		char* functype = (char *)malloc(strlen($<str>1)+strlen(funcargs)+1);
		strcpy(functype,$<str>1);
```

```
                strcpy(datatype,$<str>1);
                strcat(functype,funcargs);
                assign_datatype($<sym>2,functype);
                free(functype);
                char* funcname = strdup($<sym>2->root->id);
                last_func = lineno;
                last_fname = strdup(funcname);
                merge_table(symstack[sp],$<sym>2,1);
                push(funcname);
                free(funcname);
                merge_table(symstack[sp],$<sym>4,1);
                was_return = 0;
        }
        ;

ARGLIST
        : DATATYPE DECID {assign_datatype($<sym>2,$<str>1); $<sym>$ = $<sym>2;}
        | DATATYPE DECID COMMA ARGLIST {assign_datatype($<sym>2,$<str>1);
merge_table($<sym>2,$<sym>4,1); $<sym>$ = $<sym>2;}
        ;

EXPLIST
        : EXP {$<str>$ = strdup($<exp>1.dtype);}
        | EXP COMMA EXPLIST {
                $<str>$ = (char *)malloc(strlen($<exp>1.dtype)+1+strlen($<str>3)+1);
                strcpy($<str>$,$<exp>1.dtype);
                strcat($<str>$,",");
                strcat($<str>$,$<str>3);
        }
        ;

EXP
        : LOR_EXP
        | LOR_EXP EQ EXP {check_mod($<exp>1); check_exp($<exp>1,1,"=");
check_exp($<exp>3,1,"="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        | LOR_EXP PE EXP {check_mod($<exp>1); check_exp($<exp>1,1,"+=");
check_exp($<exp>3,1,"+="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        | LOR_EXP AE EXP {check_mod($<exp>1); check_exp($<exp>1,0,"&=");
check_exp($<exp>3,0,"&="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        | LOR_EXP XE EXP {check_mod($<exp>1); check_exp($<exp>1,0,"^=");
check_exp($<exp>3,0,"^="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        | LOR_EXP OE EXP {check_mod($<exp>1); check_exp($<exp>1,0,"|=");
check_exp($<exp>3,0,"|="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        | LOR_EXP ME EXP {check_mod($<exp>1); check_exp($<exp>1,1,"-=");
check_exp($<exp>3,1,"-="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        | LOR_EXP DE EXP {check_mod($<exp>1); check_exp($<exp>1,1,"/=");
check_exp($<exp>3,1,"/="); $<exp>$.dtype = strdup($<exp>1.dtype); $<exp>$.mod = 0;}
        ;

LOR_EXP
        : LAND_EXP
        | LOR_EXP OO LAND_EXP {check_exp($<exp>1,1,"||"); check_exp($<exp>3,1,"||");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}

LAND_EXP
```

```
	: BOR_EXP
	| LAND_EXP AA BOR_EXP {check_exp($<exp>1,1,"&&"); check_exp($<exp>3,1,"&&");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	;

BOR_EXP
	: BXOR_EXP
	| BOR_EXP '|' BXOR_EXP {check_exp($<exp>1,0,"|"); check_exp($<exp>3,0,"|");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	;

BXOR_EXP
	: BAND_EXP
	| BXOR_EXP '^' BAND_EXP {check_exp($<exp>1,0,"^"); check_exp($<exp>3,0,"^");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	;

BAND_EXP
	: REL_EXP
	| BAND_EXP '&' REL_EXP {check_exp($<exp>1,0,"&"); check_exp($<exp>3,0,"&");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	;

REL_EXP
	: CMP_EXP
	| REL_EXP EQU CMP_EXP {check_exp($<exp>1,1,"=="); check_exp($<exp>3,1,"==");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	| REL_EXP NE CMP_EXP {check_exp($<exp>1,1,"!="); check_exp($<exp>3,1,"!=");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	;

CMP_EXP
	: SH_EXP
	| CMP_EXP '<' SH_EXP {check_exp($<exp>1,1,"<"); check_exp($<exp>3,1,"<");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	| CMP_EXP '>' SH_EXP {check_exp($<exp>1,1,">"); check_exp($<exp>3,1,">");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	| CMP_EXP LE SH_EXP {check_exp($<exp>1,1,"<="); check_exp($<exp>3,1,"<=");
$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	| CMP_EXP GE SH_EXP {check_exp($<exp>1,1,">=");
 check_exp($<exp>3,1,">="); $<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	;

SH_EXP
	: AS_EXP
	| SH_EXP LS AS_EXP {check_exp($<exp>1,0,"<<"); check_exp($<exp>3,0,"<<");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	| SH_EXP RS AS_EXP {check_exp($<exp>1,0,">>"); check_exp($<exp>3,0,">>");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	;

AS_EXP
	: PDM_EXP
	| AS_EXP '+' PDM_EXP {check_exp($<exp>1,1,"+"); check_exp($<exp>3,1,"+");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
```

```
	| AS_EXP '-' PDM_EXP {check_exp($<exp>1,1,"-"); check_exp($<exp>3,1,"-");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	;

PDM_EXP
	: PRE_EXP
	| PDM_EXP '*' PRE_EXP {check_exp($<exp>1,1,"*"); check_exp($<exp>3,1,"*");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	| PDM_EXP '/' PRE_EXP {check_exp($<exp>1,1,"/"); check_exp($<exp>3,1,"/");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	| PDM_EXP '%' PRE_EXP {check_exp($<exp>1,0,"%"); check_exp($<exp>3,0,"%");
$<exp>$.dtype = iconv($<exp>1,$<exp>3); $<exp>$.mod = 0;}
	;

PRE_EXP
	: POST_EXP
	| UNARY_OPERATOR PRE_EXP {check_mod($<exp>2); $<exp>$.dtype =
strdup($<exp>2.dtype); $<exp>$.mod = 0; }
	| '+' PRE_EXP {$<exp>$.dtype = strdup($<exp>2.dtype); $<exp>$.mod = 0;}
	| '-' PRE_EXP {$<exp>$.dtype = strdup($<exp>2.dtype); $<exp>$.mod = 0;}
	| '!' PRE_EXP {$<exp>$.dtype = strdup("int"); $<exp>$.mod = 0;}
	| '~' PRE_EXP {check_exp($<exp>2,0,"~"); $<exp>$.mod = 0;}
	| '&' PRE_EXP {check_exp($<exp>2,1,"&"); check_mod($<exp>2); $<exp>$.dtype =
indirect($<exp>2.dtype); $<exp>$.mod = 0;}
	| '*' PRE_EXP {check_pnt($<exp>2.dtype); check_mod($<exp>2); $<exp>$.dtype =
deref($<exp>2.dtype); $<exp>$.mod = 1;}
	;

POST_EXP
	: ID {$<exp>$ = undeclared_identifier($<str>1);}
	| CONSTANT
	| POST_EXP UNARY_OPERATOR {check_mod($<exp>1); $<exp>$.mod = 0;}
	| ID BO EXPLIST BC {
		char *params = (char *)malloc(strlen($<str>3)+3);
		strcpy(params,"(");
		strcat(params,$<str>3);
		strcat(params,")");
		char *ftype = check_func($<str>1,params);
		undeclared_identifier($<str>1);
		$<exp>$.dtype = ftype;
		$<exp>$.mod = 0;
	}
	| ID BO BC {undeclared_identifier($<str>1); char *ftype =
check_func($<str>1,strdup("()")); $<exp>$.dtype = ftype; $<exp>$.mod = 0;}
	| POST_EXP PNT ID {check_pnt($<exp>1.dtype); $<exp>$.mod = 1;}
	| POST_EXP '.' ID {$<exp>$.mod = 1;}
	| POST_EXP '[' POST_EXP ']' {check_pnt($<exp>1.dtype); check_exp($<exp>3,0,"[]");
$<exp>$.dtype = deref($<exp>1.dtype); $<exp>$.mod = 1;}
	;
%%
void check_mod(struct exp_type exp){
	if(exp.mod&&strcmp(exp.dtype,"void"))
		return;
	printf("Semantic Error: Unmodifiable lvalue on Line %d\n",lineno);
}
```

```
struct exp_type undeclared_identifier(char * x){
        int i;
        char *scope = get_scope();
        int n = strlen(scope);
   for(i = sp; i>=0; i--){
                struct node *lu = lookup(symstack[i],x);
                int m = 0;
                if(lu)
                        m = strlen(lu->scope);
      if(lu!=NULL&&m<=n&&!strncmp(lu->scope,scope,m)){
                        struct exp_type ret;
                        ret.dtype = strdup(lu->dtype);
                        ret.mod = 1;
         return ret;
                }
   }
   printf("Semantic Warning: Undeclared Identifier %s on Line %d\n",x,lineno);
        struct exp_type ret;
        ret.dtype = strdup("void");
        ret.mod = 1;
   return ret;
}
int check_exp(struct exp_type exp, int am, char *op){
        for(int i = 0; i<7; i++){
                if(!strcmp(exp.dtype,allow_types[i])){
                        if(allow_mask[am][i])
                                return 1;
                        else{
                                printf("Semantic Error: ");
                                if(strlen(exp.dtype)<=3)
                                        printf("Operator ");
                                printf("%s not defined on type %s on Line
%d\n",op,exp.dtype,lineno);
                                return 0;
                        }
                }
        }
        printf("Semantic Error: Operator %s not defined on type %s on Line
%d\n",op,exp.dtype,lineno);
        return 0;
}
char *iconv(struct exp_type e1, struct exp_type e2){
        int t1 = -1,t2 = -1;
        for(int i = 0; i<7; i++)
                if(!strcmp(e1.dtype,allow_types[i]))
                 t1 = i;
        for(int i = 0; i<7; i++)
                if(!strcmp(e2.dtype,allow_types[i]))
                 t2 = i;
        int t = (t1<t2)?(t2):(t1);
        return strdup(allow_types[t]);
}
int main(){
   init();
   yyin = fopen("test/test_file.c","r");
```

```
    yyparse();
        if(last_func == -1)
            printf("Semantic Warning: No functions found\n");
        else if(strcmp(last_fname,"main"))
            printf("Semantic Warning: Main was not the last function declared, last function was
%s on Line %d\n",last_fname,last_func);
        printsymboltable();
    return 0;
}
int yyerror(const char *s){
    return printf("Syntax Error: Line %d\n",lineno);
}
int yywrap(){
    return 1;
}
```

Code for Symbol Table:

```
#include "symboltable.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int lineno;

struct symboltable *createsymboltable(){
    struct symboltable *ret = (struct symboltable *)malloc(sizeof(struct symboltable));
    ret->root = NULL;
    return ret;
}

void add_identifier(struct symboltable *st, char *id, char *dtype){
    if((st->root)==NULL){
        st->root = (struct node *)malloc(sizeof(struct node));
        (st->root)->id = strdup(id);
        (st->root)->dtype = strdup(dtype);
        (st->root)->line = lineno;
        (st->root)->scope = strdup("");
        (st->root)->next = NULL;
    }else{
        struct node* focus = st->root;
        while(focus->next) focus = focus->next;
        focus->next = (struct node *)malloc(sizeof(struct node));
        (focus->next)->id = strdup(id);
        (focus->next)->dtype = strdup(dtype);
        (focus->next)->line = lineno;
        (focus->next)->scope = strdup("");
        (focus->next)->next = NULL;
    }
}

struct node* lookup(struct symboltable *st, char *key){
    if(key==NULL)
        key = strdup("");
    struct node *focus = st->root;
    while(focus!=NULL){
        if(!strcmp(key,focus->id))
            return focus;
        focus=(focus->next);
    }
    return NULL;
}

void apply_scope(struct symboltable *st, char *scope){
    struct node *focus = st->root;
    while(focus){
        char *newscope = (char *)malloc(strlen(scope)+1+strlen(focus->scope));
        strcpy(newscope,scope);
        if(strlen(focus->scope)){
            strcat(newscope,".");
            strcat(newscope,focus->scope);
```

```
        }
        free(focus->scope);
        focus->scope = newscope;
        focus = focus->next;
    }
}

void merge_table(struct symboltable* dest, struct symboltable *src, int check_conflict){
    struct node* focus = dest->root;
    if(focus==NULL)
        dest->root = src->root;
    else{
        while(focus->next) focus = focus->next;
        struct node *ins = src->root;
        while(ins){
            if(check_conflict){
                struct node* lu = lookup(dest,ins->id);
                if(lu&&!strcmp(ins->scope,lu->scope)){
                    printf("Semantic Warning: Redefinition of identifier %s on Line %d\n",ins->id,ins-
>line);
                }else{
                    focus->next = (struct node *)malloc(sizeof(struct node));
                    (focus->next)->id = strdup(ins->id);
                    (focus->next)->dtype = strdup(ins->dtype);
                    (focus->next)->line = ins->line;
                    (focus->next)->scope = strdup(ins->scope);
                    (focus->next)->next = NULL;
                    focus = focus->next;
                }
            }else{
                focus->next = (struct node *)malloc(sizeof(struct node));
                (focus->next)->id = strdup(ins->id);
                (focus->next)->dtype = strdup(ins->dtype);
                (focus->next)->line = ins->line;
                (focus->next)->scope = strdup(ins->scope);
                (focus->next)->next = NULL;
                focus = focus->next;
            }
            ins = ins->next;
        }
        focus->next = NULL;
    }
}

void assign_datatype(struct symboltable *st, char *dtype){
    struct node* focus = st->root;
    while(focus){
        char *newdtype = (char *)malloc(strlen(focus->dtype)+strlen(dtype));
        strcpy(newdtype,dtype);
        focus->dtype[strlen(focus->dtype)-1] = '\0';
        strcat(newdtype,focus->dtype);
        free(focus->dtype);
        focus->dtype = newdtype;
        focus = focus->next;
    }
```

```
}

void apply_indirection(struct symboltable *st){
    struct node* focus = st->root;
    while(focus){
        char *newdtype = (char *)malloc(strlen(focus->dtype)+2);
        newdtype[0] = '\0';
        strcat(newdtype,"*");
        strcat(newdtype,focus->dtype);
        free(focus->dtype);
        focus->dtype = newdtype;
        focus = focus->next;
    }
}

char *dstring(struct symboltable *st){
    struct node *focus = st->root;
    int len = 0;
    while(focus){
        len+=strlen(focus->dtype);
        len++;
        focus = focus->next;
    }
    if(!len)
        return strdup("");
    char *ans = (char *)malloc(len+2);
    strcpy(ans,"(");
    focus = st->root;
    while(focus){
        strcat(ans,focus->dtype);
        strcat(ans,",");
        focus=focus->next;
    }
    ans[len] = ')';
    ans[len+1] = '\0';
    return ans;
}

void printsym(struct symboltable *st){
    struct node *focus = st->root;
    while(focus){
        printf("%s\t%d\n",focus->id,focus->line);
        focus = focus->next;
    }
}
```

Code for header file of the symbol table:

```
struct node{
    char *id, *dtype;
    char *scope;
    int line;
    struct node* next;
};

struct symboltable{
    struct node* root;
};
struct symboltable *createsymboltable();
void add_identifier(struct symboltable* st, char* id, char* dtype);
void apply_scope(struct symboltable* st, char* scope);
struct node* lookup(struct symboltable* st, char *key);
void merge_table(struct symboltable* dest, struct symboltable *src, int check_conflict);
void assign_datatype(struct symboltable* st, char *dtype);
void apply_indirection(struct symboltable* st);
void printsym(struct symboltable* st);
char *dstring(struct symboltable *st);
```

Test Cases And Screenshots:

Test 1

```
/* test case to check for variable declarations */
int main(int a){
	int a;
	int b;
	int abc;
	// will give a warning for undeclared identifier c
	a = b + c;

	if(a){
		if(b){
			if(a)
				int x,y;
			int v;
		}
		int u;
	}
	int z;
	// will give a warning for redeclared variable
	int a;

}
```

```
Semantic Warning: Redefinition of identifier a on Line 3
Semantic Warning: Undeclared Identifier c on Line 7
Semantic Error: + not defined on type void on Line 7
Semantic Error: = not defined on type void on Line 7
Semantic Warning: Redefinition of identifier a on Line 19
Semantic Warning No return found in function of type int on Line 21
Symbol Table
```

Identifier Name	Line	Datatype	Scope
main	2	int(int)	
a	2	int	main
b	4	int	main
abc	5	int	main
x	12	int	main.if.if.if
y	12	int	main.if.if.if
v	13	int	main.if.if
u	15	int	main.if
z	17	int	main

Test 2

```
// test case to check for errors regarding operator

int main(){

        // will give an error since not modifiable
        5++;

        float a;

        //will give an error since operator is not defined for float operands
        ~a;

        float b;
        int k;
        a+b = 5;

        // will give an error since logical operator are not defined for float operands
        k = a&b;

}
```

```
Semantic Error: Unmodifiable lvalue on Line 6
Semantic Error: ~ not defined on type float on Line 11
Semantic Error: Unmodifiable lvalue on Line 15
Semantic Error: & not defined on type float on Line 18
Semantic Error: & not defined on type float on Line 18
Semantic Warning No return found in function of type int on Line 20
Symbol Table
-------------------------------------------------------------------------------------------
|   Identifier Name     |   Line                |   Datatype            |   Scope               |
-------------------------------------------------------------------------------------------
|   main                |   3                   |   int()               |                       |
-------------------------------------------------------------------------------------------
|   a                   |   8                   |   float               |   main                |
-------------------------------------------------------------------------------------------
|   b                   |   13                  |   float               |   main                |
-------------------------------------------------------------------------------------------
|   k                   |   14                  |   int                 |   main                |
-------------------------------------------------------------------------------------------
```

Test 3

```
// test case to check error regarding referencing of a pointer and some array errors

void main(){

        int a;

        // will give an error since a is of type int an * expects a type pointer
        *a;

        int *b;

        //will not give an error since b is defined as int*
        int c = -*b;

        int **aa;

        //will give an error since a is defined only as a[][]
        aa[2][2][2];


        //will give an error
        ab[1.1];

        return 1;

}
```

```
Semantic Error: Expression requires pointer type, int received on Line 8
Semantic Error: Expression requires pointer type, int received on Line 18
Semantic Warning: Undeclared Identifier ab on Line 22
Semantic Error: Expression requires pointer type, void received on Line 22
Semantic Error: [] not defined on type float on Line 22
Semantic Warning Found return in function of type void on Line 26
Symbol Table
```

Identifier Name	Line	Datatype	Scope
main	3	void()	
a	5	int	main
b	10	int*	main
c	13	int	main
aa	15	int**	main

Test 4

```
// test case to show the implementation of scope
int max(int a, int b)
{
	a = b;
}

int main(){
	int a;
	int b;
	a = b;
}
```

```
Semantic Warning No return found in function of type int on Line 5
Semantic Warning No return found in function of type int on Line 11
Symbol Table
```

Identifier Name	Line	Datatype	Scope
max	2	int(int,int)	
a	2	int	max
b	2	int	max
main	7	int()	
a	8	int	main
b	9	int	main

Test 5

```
// test case to check errors regarding functions


int maxval(int a, int b)
        return a;
}

int main()

        int a;

        //will give an error since maxval expects an (int, int) as parameters but it is getting (float,
int)
        a = maxval(1.0, 1);

        int b ;

        //will give an error saying that maxval expects an (int, int) as parameters but it is getting
(int, int, int)
        b = maxval(1,1,1);

        // error saying no return statement
}
```

```
Syntax Error: Line 5
Semantic Warning: Main was not the last function declared, last function was maxval on Line 4
Symbol Table
-------------------------------------------------------------------------------------------------
|   Identifier Name     |   Line                |   Datatype            |   Scope               |
-------------------------------------------------------------------------------------------------
|   maxval              |   4                   |   int(int,int)        |                       |
-------------------------------------------------------------------------------------------------
```

Test 6

```
// test case to check if main function is last func or not

int main(){

	int a = 1;
	return a;
}

int maxval(){
	int b ;
	b= 1;
	return b;
}
```

```
Semantic Warning: Main was not the last function declared, last function was maxval on Line 9
Symbol Table
-------------------------------------------------------------------------------------------------
|   Identifier Name     |   Line                |   Datatype            |   Scope               |
-------------------------------------------------------------------------------------------------
|   main                |   3                   |   int()               |                       |
-------------------------------------------------------------------------------------------------
|   a                   |   5                   |   int                 |   main                |
-------------------------------------------------------------------------------------------------
|   maxval              |   9                   |   int()               |                       |
-------------------------------------------------------------------------------------------------
|   b                   |   10                  |   int                 |   maxval              |
-------------------------------------------------------------------------------------------------
```

Conclusion:

We build upon the syntax analyzer presented in the last phase of the project to produce a miniature C compiler with an additional semantic analyzer. The input source program that passes through our program error free is expected to be semantically as well as syntactically correct. We improve upon the parser by removing conflicts and making the symbol table more robust and feature-complete. The addition of semantic warnings and errors give valuable information on how to improve an incorrect program at a high level. The two most important additions, the expression evaluator and the scoped symbol table tree, are easy to extend to the entirety of the ANSI C standard; we have already included support for all binary and unary operators as well as all datatypes, both simple and compound offered by any major C compiler available today (gcc, Visual C++, etc.). We will continue to build upon this foundation for the next phase of the project and hope to have a compiler that meets the full ANSI C specification by time of completion.