Semantic Analyser and Intermediate Code Generation for C

Anagh Singh - 13CO110
Aishwarya Rajan - 13CO109

April 29, 2016

Compiler Design Lab - CO351

Department of Computer Science and Engineering, NITK

Abstract

Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known. In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking. The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.

Most compilers translate the source program first to some form of intermediate representation and convert from there into machine code. The intermediate representation is a machine- and language- independent version of the original source code.

Contents

Listings

1 Introduction

Semantic analysis, also context sensitive analysis, is a process in compiler construction after parsing to gather necessary semantic information from the source code. It is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known. In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking. The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.

Most compilers translate the source program first to some form of intermediate representation and convert from there into machine code. The intermediate representation is a machine- and language- independent version of the original source code.

Although converting the code twice introduces another step, use of an intermediate representation provides advantages in increased abstraction, cleaner separation between the front and back ends, and adds possibilities for re-targeting/cross-compilation. Intermediate representations also lend themselves to supporting advanced compiler optimizations and most optimization is done on this form of the code.

There are many intermediate representations in use but the various representations are actually more alike than they are different. Intermediate representations are usually categorized according to where they fall between a high-level language and machine code. IRs that are close to a high-level language are called high-level IRs, and IRs that are close to assembly are called low-level IRs. For example, a high-level IR might preserve things like array subscripts or field accesses whereas a low-level IR converts those into explicit addresses and offsets.

Intermediate codes are machine independent codes, but they are close to machine instructions. The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator. Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language. In our project, we have used three-address code (quadruples) as the intermediate language.

1.1 Symbol Table

A symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location. Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination. Now well move forward to semantic analysis, where we delve even deeper to check whether they form a sensible set of instructions in the programming language.

Whereas any old noun phrase followed by some verb phrase makes a syntactically correct English sentence, a semantically correct one has subject-verb agreement, proper use of gender, and the components go together to express an idea that makes sense. For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth. Semantic analysis is the front ends penultimate phase and the compilers last chance to weed out incorrect programs. We need to ensure the program is sound enough to carry on to code generation.

A large part of semantic analysis consists of tracking variable/function/type declarations and type checking. In many languages, identifiers have to be declared before theyre used. As the compiler encounters a new declaration, it records the type information assigned to that identifier. Then, as it continues examining the rest of the program, it verifies that the type of an identifier is respected in terms of the operations being performed. For example, the type of the right side expression of an assignment statement should match the type of the left side, and the left side needs to be a properly declared and assignable identifier.

The parameters of a function should match the arguments of a function call in both number and type. The language may require that identifiers be unique, thereby forbidding two global declarations from sharing the same name. Arithmetic operands will need to be of numericperhaps even the exact same type (no automatic int-to-double conversion, for instance). These are examples of the things checked in the semantic analysis phase. Some semantic analysis might be done right in the middle of parsing.

As a particular construct is recognized, say an addition expression, the parser action could check the two operands and verify they are of numeric type and compatible for this operation. In fact, in a one-pass compiler, the code is generated right then and there as well. In a compiler that runs in more than one pass the first pass digests the syntax and builds a parse tree representation of the program. A second pass traverses the tree to verify that the program respects all semantic rules as well. The single-pass strategy is typically more efficient, but multiple passes allow for better modularity and flexibility (i.e., can often order things arbitrarily in the source program).

1.2 Three-Address Code

Three-address code (often abbreviated to TAC or 3AC) is a form of representing intermediate code used by compilers to aid in the implementation of code-improving transformations. Each instruction in three-address code can be described as a 4-tuple: (operator, operand1, operand2, result). In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like $x + y * z$ might be translated into the sequence of three-address instructions:

```
t1= y * z
t2= x + t1
```

Here, $t1$ and $t2$ are compiler-generated temporary names. This unravelling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

The key features of three-address code are that every instruction implements exactly one fundamental operation, and that the source and destination may refer to any available register.

1.3 Types of 3AC Statements

Types of Three-Address Statements used:-

1. Assignment statements:-

   ```
   x=y op z
   ```

   where *op* is binary operator.

2. Assignment instructions:-

   ```
   x=op y
   ```

   where *op* is unary operator.

3. Procedure calls and parameters:-

   ```
   Param x1
   Param x2

   Param xn
   Call p,n
   ```

   Here *param*s are parameters, x_is are values passed, p is the procedure name and n is number of parameters passed.

4. Indexed Assignment:-

   ```
   x=y[i]+j
   ```

   $y[i]$ usual array notation.

5. Address and Pointer assignments:-

   ```
   x=&y
   x=*y
   ```

   & and * are address of and value at C notations.

6. Structure and union variables:-

```
a.c=b.c*c;
```

a and b are structures and c is part of structure.

```
p->a=c+b;
```

, the usual C notation.

1.4 The YACC Tool

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low- level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is developed for the Unix operating system. The name is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF. It was developed in 1970 by Stephen C. Johnson at AT&T Corporation and originally written in the B programming language. Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper.

The tool can be decomposed into the following:

1. Working
   Yacc turns the specification file into a C program, which parses the input according to the specification given. The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read. The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

   (a) Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

(b) Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

   i. The shift action is the most common action the parser takes.

   ii. The reduce action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

   iii. Accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job.

   iv. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing.

2. Precedence Rules The precedences and associativities are used by Yacc to resolve parsing conflicts; they give riseto disambiguating rules. Formally, the rules work as follows:

   (a) The precedences and associativities are recorded for those tokens and literals that have them.

   (b) A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the is used it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

   (c) When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

   (d) If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and non-associating implies error.

1.5 The YACC Script

Yacc is a computer program for the Unix operating system. It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF. Yacc itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted as the default by more recent, largely compatible, programs.

YACC is an acronym for "Yet Another Compiler Compiler". It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF.

The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules. Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees.

Yacc produces only a parser (phrase analyzer); for full syntactic analysis this requires an external lexical analyzer to perform the first tokenization stage (word analysis), which is then followed by the parsing stage proper. Lexical analyzer generators, such as Lex or Flex are widely available. The IEEE POSIX P1003.2 standard defines the functionality and requirements for both Lex and Yacc.

1.5.1 Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)
In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

1.5.2 Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces “” and “”. For example,

```
A       :       '('  B  ')'
                         {       hello( 1, "abc" );  }
```

and

```
XXX     :       YYY  ZZZ
                         {       printf("a message\n");
                                 flag = 25;   }
```

are grammar rules with actions.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks “%” and “%”. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
\%{   int variable = 0;   \%}
```

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in “yy”; we should avoid such names.

1.5.3 Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the token number, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the “# define” mechanism of C is used to allow the lexical analyzer to return these numbers symbolically.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

1.5.4 Parsing

Yacc turns the specification file into a C program, which parses the input according to the specification given. The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

- Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.
- Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable yylval is copied onto the value stack. After the return from the user code, the reduction is carried out. When the goto action is done, the external variable yyval is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The error action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input.

1.5.5 Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr    :       expr  '-'  expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two disambiguating rules by default:

- In a shift/reduce conflict, the default is to do the shift.
- In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

1.5.6 Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line

are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
\%left  '+'  '-'
\%left  '*'  '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative.

1.5.7 Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output. It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted. Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc

1.5.8 YACC Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called y.tab.c on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called yyparse; it is an integer valued function. When it is called, it in turn repeatedly calls yylex, the lexical analyzer supplied by the user to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) yyparse returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, yyparse returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called main must be defined, that eventually calls yyparse. In addition, a routine called yyerror prints a message when a syntax error is detected.

The argument to yyerror is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable yychar contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the main program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable yydebug is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

1.6 Abstract Syntax Tree

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construct like an if-condition-then expression may be denoted by means of a single node with two branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are often built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis. Abstract syntax trees are also used in program analysis and program transformation systems.

Abstract syntax trees are data structures widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

Being the product of the syntax analysis phase of a compiler, the AST has a few properties that are invaluable to the further steps of the compilation process. When compared to the source code, an AST does not include certain elements, such as inessential punctuation and delimiters (braces, semicolons, parentheses, etc.). A more important difference is that the AST can be edited and enhanced with properties and annotations for every element it contains. Such editing and annotation is impossible with the source code of a program, since it would imply changing it.

2 Design of Program

2.1 Code for Scanner

This section contains our actual scanner. The code for it is:

```
D           [0-9]
L           [a-zA-Z_]
H           [a-fA-F0-9]
```

```
E           [Ee][+-]?{D}+
FS          (f|F|l|L)
IS          (u|U|l|L)*

%{
#include <stdio.h>
#include "y.tab.h"
int cnt=1;
int line=1;
char tempid[100];
%}

%%
"/*"            {comment();}
"printf"        { cnt+=yyleng;ECHO; return(PRINTF); }
"scanf"         { cnt+=yyleng;ECHO; return(SCANF); }
"auto"          { cnt+=yyleng;ECHO; return(AUTO); }
"break"         { cnt+=yyleng;ECHO; return(BREAK); }
"case"          { cnt+=yyleng;ECHO; return(CASE); }
"char"          { cnt+=yyleng;ECHO; return(CHAR); }
"const"         { cnt+=yyleng;ECHO; return(CONST); }
"continue"      { cnt+=yyleng;ECHO; return(CONTINUE); }
"default"       { cnt+=yyleng;ECHO; return(DEFAULT); }
"do"            { cnt+=yyleng;ECHO; return(DO); }
"double"        { cnt+=yyleng;ECHO; return(DOUBLE); }
"else"          { cnt+=yyleng;ECHO; return(ELSE); }
"enum"          { cnt+=yyleng;ECHO; return(ENUM); }
"extern"        { cnt+=yyleng;ECHO; return(EXTERN); }
"float"         { cnt+=yyleng;ECHO; return(FLOAT); }
"for"           { cnt+=yyleng;ECHO; return(FOR); }
"goto"          { cnt+=yyleng;ECHO; return(GOTO); }
"if"            { cnt+=yyleng;ECHO; return(IF); }
"int"           { cnt+=yyleng;ECHO; return(INT); }
"long"          { cnt+=yyleng;ECHO; return(LONG); }
"register"      { cnt+=yyleng;ECHO; return(REGISTER); }
"return"        { cnt+=yyleng;ECHO; return(RETURN); }
"short"         { cnt+=yyleng;ECHO; return(SHORT); }
"signed"        { cnt+=yyleng;ECHO; return(SIGNED); }
"sizeof"        { cnt+=yyleng;ECHO; return(SIZEOF); }
"static"        { cnt+=yyleng;ECHO; return(STATIC); }
"struct"        { cnt+=yyleng;ECHO; return(STRUCT); }
"switch"        { cnt+=yyleng;ECHO; return(SWITCH); }
"typedef"       { cnt+=yyleng;ECHO; return(TYPEDEF); }
"union"         { cnt+=yyleng;ECHO; return(UNION); }
"unsigned"      { cnt+=yyleng;ECHO; return(UNSIGNED); }
"void"          { cnt+=yyleng;ECHO; return(VOID); }
"volatile"      { cnt+=yyleng;ECHO; return(VOLATILE); }
"while"         { cnt+=yyleng;ECHO; return(WHILE); }
(['])+({L}|{D})+(['])           { cnt+=yyleng;ECHO; return(SINGLE); }
{L}({L}|{D})*       { cnt+=yyleng;ECHO; strcpy(tempid,yytext);return(IDENTIFIER); }

0[xX]{H}+{IS}?      { cnt+=yyleng;ECHO; return(CONSTANT); }
0{D}+{IS}?      { cnt+=yyleng;ECHO; return(CONSTANT); }
{D}+{IS}?       { cnt+=yyleng;ECHO; return(CONSTANT); }
L?'(\\.|[^\\'])+'   { cnt+=yyleng;ECHO; return(CONSTANT); }

{D}+{E}{FS}?        { cnt+=yyleng;ECHO; return(CONSTANT); }
{D}*"."{D}+({E})?{FS}?  { cnt+=yyleng;ECHO; return(CONSTANT); }
{D}+"."{D}*({E})?{FS}?  { cnt+=yyleng;ECHO; return(CONSTANT); }

L?\"(\\.|[^\\"])*\" { cnt+=yyleng;ECHO; return(STRING_LITERAL); }

"..."           { cnt+=yyleng;ECHO; return(ELLIPSIS); }
">>="           { cnt+=yyleng;ECHO; return(RIGHT_ASSIGN); }
"<<="           { cnt+=yyleng;ECHO; return(LEFT_ASSIGN); }
"+="            { cnt+=yyleng;ECHO; return(ADD_ASSIGN); }
"-="            { cnt+=yyleng;ECHO; return(SUB_ASSIGN); }
"*="            { cnt+=yyleng;ECHO; return(MUL_ASSIGN); }
"/="            { cnt+=yyleng;ECHO; return(DIV_ASSIGN); }
```

```
"%="            { cnt+=yyleng;ECHO; return(MOD_ASSIGN); }
"&="            { cnt+=yyleng;ECHO; return(AND_ASSIGN); }
"^="            { cnt+=yyleng;ECHO; return(XOR_ASSIGN); }
"|="            { cnt+=yyleng;ECHO; return(OR_ASSIGN); }
">>"            { cnt+=yyleng;ECHO; return(RIGHT_OP); }
"<<"            { cnt+=yyleng;ECHO; return(LEFT_OP); }
"++"            { cnt+=yyleng;ECHO; return(INC_OP); }
"--"            { cnt+=yyleng;ECHO; return(DEC_OP); }
"->"            { cnt+=yyleng;ECHO; return(PTR_OP); }
"&&"            { cnt+=yyleng;ECHO; return(AND_OP); }
"||"            { cnt+=yyleng;ECHO; return(OR_OP); }
"<="            { cnt+=yyleng;ECHO; return(LE_OP); }
">="            { cnt+=yyleng;ECHO; return(GE_OP); }
"=="            { cnt+=yyleng;ECHO; return(EQ_OP); }
"!="            { cnt+=yyleng;ECHO; return(NE_OP); }
";"         { cnt+=yyleng;ECHO; return(';'); }
("{"|"<%")      { cnt+=yyleng;ECHO; return('{'); }
("}"|"%>")      { cnt+=yyleng;ECHO; return('}'); }
","         { cnt+=yyleng;ECHO; return(','); }
":"         { cnt+=yyleng;ECHO; return(':'); }
"="         { cnt+=yyleng;ECHO; return('='); }
"("         { cnt+=yyleng;ECHO; return('('); }
")"         { cnt+=yyleng;ECHO; return(')'); }
("["|"<:")      { cnt+=yyleng;ECHO; return('['); }
("]"|":>")      { cnt+=yyleng;ECHO; return(']'); }
"."         { cnt+=yyleng;ECHO; return('.'); }
"&"         { cnt+=yyleng;ECHO; return('&'); }
"!"         { cnt+=yyleng;ECHO; return('!'); }
"~"         { cnt+=yyleng;ECHO; return('~'); }
"-"         { cnt+=yyleng;ECHO; return('-'); }
"+"         { cnt+=yyleng;ECHO; return('+'); }
"*"         { cnt+=yyleng;ECHO; return('*'); }
"/"         { cnt+=yyleng;ECHO; return('/'); }
"%"         { cnt+=yyleng;ECHO; return('%'); }
"<"         { cnt+=yyleng;ECHO; return('<'); }
">"         { cnt+=yyleng;ECHO; return('>'); }
"^"         { cnt+=yyleng;ECHO; return('^'); }
"|"         { cnt+=yyleng;ECHO; return('|'); }
"?"         { cnt+=yyleng;ECHO; return('?'); }

[ ]         {cnt+=yyleng;ECHO;}
[\t\v\f]        { cnt+=yyleng; }
[\n]            {line++;cnt=1;}
.           { /* ignore bad characters */ }

%%
yywrap()
{
    return(1);
}
comment()
{
    char c, c1;
loop:
    while ((c = input()) != '*' && c != 0)
    {
        if(c=='\n') {line++;cnt=1;}
        else    {cnt++;}
    }
        //putchar(c); PUTCHAR only if comments need to be shown!
    if ((c1 = input()) != '/' && c1 != 0)
    {
        unput(c1);
        goto loop;
    }
}
```

2.2 Semantic Parser

This section contains our actual parser. The code for it is:

```
%{
#include <stdio.h>
#include <string.h>
#include "symbol_table.h"
extern FILE *yyin;
extern FILE *yyout;
extern int column;
extern int line;
extern int cnt;
extern char *yytext,tempid[100];
int temp,err,err1=0;

install()
{
    symrec *s;
    s = getsym (tempid);
    if (s == 0)
    s = putsym (tempid,temp);
    else
    {
//        printf(" VOID=1 ");
//    printf(" CHAR=2 ");
//    printf(" INT=3 ");
//    printf(" FLOAT=4 ");
//    printf(" DOUBLE=4 ");
        printf("
            ********************************************************************************************
            n");
        printf( "\nA Semantic error has been encountered at Pos : %d : %d : %s is already
            defined as %d\n\n",line,cnt,s->name,s->type );
        printf("
            ********************************************************************************************
            n");
//        exit(0);
    }
    err1=1;
}
int context_check()
{
    symrec *s;
    s = getsym(tempid);
    if (s == 0 )

    {
     printf("
          ********************************************************************************************
          n");
    printf( "\nA Semantic error has been encountered at Pos : %d : %d : %s is an
        undeclared identifier\n\n",line,cnt,tempid);//exit(0);
 printf("
     ********************************************************************************************
     n");

    return 0;}
    else
    return(s->type);
    err1=1;

}
type_err(int t1,int t2)
{
    if(t1&&t2)
    {
//    printf(" VOID=1 ");
//    printf(" CHAR=2 ");
```

```
//    printf(" INT=3 ");
//    printf(" FLOAT=4 ");
//    printf(" DOUBLE=4 ");
      printf("****************************************************************************************************
        n");

     printf( "\nA Semantic error was encountered at Pos : %d : %d : Type mismatch for %s
        between %d and %d \n\n",line,cnt,tempid,t1,t2);
      printf("****************************************************************************************************
        n");

     err1=1;
//   exit(0);
     }
}

%}



%token 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 SINGLE PRINTF SCANF

%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 LOWER_THAN_ELSE
%nonassoc ELSE

%start translation_unit
%%

primary_expression
     : IDENTIFIER     {$$=context_check();}
     | CONSTANT
     | STRING_LITERAL
     | '(' expression ')' {$$= $2;}
     ;

postfix_expression
     : primary_expression     {$$=$1;}
     | postfix_expression '[' expression ']'
     | postfix_expression '(' ')'
     | postfix_expression '(' argument_expression_list ')'
     | postfix_expression '.' IDENTIFIER
     | postfix_expression PTR_OP IDENTIFIER
     | postfix_expression INC_OP
     | postfix_expression DEC_OP
     ;

argument_expression_list
     : assignment_expression
     | argument_expression_list ',' assignment_expression
     ;

unary_expression
     : postfix_expression     {$$=$1;}
     | INC_OP unary_expression
     | DEC_OP unary_expression
     | unary_operator cast_expression
     | SIZEOF unary_expression
     | SIZEOF '(' type_name ')'
```

```
    ;

unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;

cast_expression
    : unary_expression  {$$=$1;}
    | '(' type_name ')' cast_expression
    ;

multiplicative_expression
    : cast_expression   {$$=$1;}
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;

additive_expression
    : multiplicative_expression {$$=$1;}
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

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
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

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 inclusive_or_expression
    ;

```

```
logical_or_expression
    : logical_and_expression    {$$=$1;}
    | logical_or_expression OR_OP logical_and_expression
    ;

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

assignment_expression
    : conditional_expression    {$$=$1;}
    | unary_expression assignment_operator assignment_expression    {if($1!=$3){type_err($1,$3);}}
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

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

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
```

```
    ;

type_specifier
    : VOID   {temp=1;}
    | CHAR   {temp=2;}
    | SHORT {temp=3;}
    | INT     {temp=3;}
    | LONG   {temp=3;}
    | FLOAT {temp=4;}
    | DOUBLE      {temp=4;}
    | SIGNED
    | UNSIGNED
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'     {install();}
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER     {install();}
    ;

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
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

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     {context_check();}
    | IDENTIFIER '=' constant_expression     //{context_check();}
```

```
    ;

type_qualifier
    : CONST
    | VOLATILE
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER    {install();}
    | '(' declarator ')'
    | direct_declarator '[' constant_expression ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;

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

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;


parameter_type_list
    : parameter_list
    | parameter_list ',' ELLIPSIS
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER    {install();}
    | identifier_list ',' IDENTIFIER    {install();}
    ;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;

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
    ;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

labeled_statement
    : IDENTIFIER ':' statement  //{context_check();}
    | CASE constant_expression ':' statement
    | DEFAULT ':' statement
    ;

compound_statement
    : '{' '}'
    | '{' statement_list '}'
    | '{' declaration_list '}'
    | '{' declaration_list statement_list '}'
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;

statement_list
    : statement
    | statement_list statement
    ;

expression_statement
    : ';'
    | expression ';'
    ;

selection_statement
    : IF '(' expression ')' statement  %prec LOWER_THAN_ELSE ;

    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
```

```
	| FOR '(' expression_statement expression_statement expression ')' statement
	;

jump_statement
	: GOTO IDENTIFIER ';'   //{context_check();}
	| CONTINUE ';'
	| BREAK ';'
	| RETURN ';'
	| RETURN expression ';'
	;

translation_unit
	: external_declaration
	| translation_unit external_declaration
	;

external_declaration
	: function_definition
	| declaration
	;

function_definition
	: declaration_specifiers declarator declaration_list compound_statement
	| declaration_specifiers declarator compound_statement
	| declarator declaration_list compound_statement
	| declarator compound_statement
	;

declaration : error ';'
%%

yyerror(s)
char *s;
{
	fflush(stdout);err=1;
	printf("Syntax error at Pos : %d : %d\n",line,cnt);
	exit(0);
	//printf("\n%*s\n%*s\n", column, "^", column, s);
}
main(argc,argv)
int argc;
char **argv;
{

	char *fname;
	++argv,--argc;/*skip program name*/
	if(argc>0)
	{
		yyin=fopen(argv[0],"r");
		fname=argv[0];
		strcat(fname,"_program");
		yyout=fopen(fname,"w");
	}
	else
	{
		printf("Please give the c filename as an argument.\n");
	}
	yyparse();
	if(err==0)
	printf("No Syntax errors found!\n");
	fname=argv[0];strcat(fname,"_table");
	FILE *sym_tab=fopen(fname,"w");
	fprintf(sym_tab,"Type\tSymbol\n");
	symrec *ptr;
	for(ptr=sym_table;ptr!=(symrec *)0;ptr=(symrec *)ptr->next)
	{
		fprintf(sym_tab,"%d\t%s\n",ptr->type,ptr->name);
	}
	fclose(sym_tab);
```

```
}
```

2.3 Code for Symbol Table

This section contains the code for producing the symbol table from the given program. The code for it is:

```
#define t_void  1
#define t_char  2
#define t_int   3
#define t_float 4
struct symrec
{
    char *name;
    int type;
    struct symrec *next;
};
typedef struct symrec symrec;
symrec *sym_table = (symrec *)0;
symrec *putsym();
symrec *getsym();
symrec *putsym(char *sym_name,int sym_type)
{
    symrec *ptr;
    ptr=(symrec *)malloc(sizeof(symrec));
    ptr->name=(char *)malloc(strlen(sym_name)+1);
    strcpy(ptr->name,sym_name);
    ptr->type=sym_type;
    ptr->next=(struct symrec *)sym_table;
    sym_table=ptr;
    return ptr;
}
symrec *getsym(char *sym_name)
{
    symrec *ptr;
    for(ptr=sym_table;ptr!=(symrec *)0;ptr=(symrec *)ptr->next)
    if(strcmp(ptr->name,sym_name)==0)
    return ptr;
    return 0;
}
```

The functions defined in this header file are used to not only implement the symbol table but semantic checking too is implemented using them appropriately. It is explicitly clear in the yacc file functions where they are used to type check, check for multiple declarations and etc.

2.4 Code Generation Parser

This section contains our actual parser. The code for it is:

```
%{

// Linux:   flex quadComp.l; bison -d -b y quadComp.y; flex quadComp.l; gcc global.c lex.yy.c y.tab.c -lfl -lm -o quad; cat input.c | ./quad
// Windows: flex quadComp.l; bison -d -b y quadComp.y; flex quadComp.l; /cygdrive/c/cygwin/bin/gcc.exe global.c lex.yy.c y.tab.c -lfl -lm -o quad.exe; ./quad.exe < input.c > output.out


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

```c

//Prologue
void yyerror(char * message);

int rel_addr = 0; // relative (to ebp) addr of current local variable
symtabEntry *current_function = NULL;
int instructionCounter = 0;

char instructions[10000][1000];

int get_type_size(symtabEntryType type) {
    return type == INTEGER ? 4 : 8;    // int and real are of the size 4
}

void patch(int ic) {
    sprintf(instructions[ic],"%s %d",instructions[ic],instructionCounter);
}

symtabEntry* declare_function(char* name, symtabEntryType returnType) {
    symtabEntry* f = lookup(name);
    if(f){
        if(f->internType != returnType) {
            yyerror("Function defined twice with differing return type.");
        }
    }
    else {
        f = addSymboltableEntry(theSymboltable, name, FUNC, returnType, 0, 0, 0, 0, 0, -1);
    }
    return f;
}

symtabEntry* variable_lookup(char* id){
    symtabEntry* e = lookup(id);
    return (e != NULL && e->vater == current_function) ? e : NULL;
}

symtabEntry* create_variable(symtabEntryType type, char* name) {
    if(variable_lookup(name)) {
        yyerror("Variable defined twice.");
    }

    symtabEntry* entry = addSymboltableEntry(theSymboltable, name, type, NOP, rel_addr, 0, 0, 0, current_function, 0);
    rel_addr += get_type_size(type);
    return entry;
}

symtabEntry* create_helper_variable(symtabEntryType type) {
    static help_num = 0;    // current helper variable number

    char* str = malloc(1000);

    sprintf(str,"V__H%d",help_num);

    ++help_num;

    return create_variable(type,str);
}

symtabEntry* create_parameter(symtabEntryType type, char* name, int param) {
    // we ignore the fact that parameters in declarations may have
    // differing or no names.

    symtabEntry* v = variable_lookup(name);
    if(!v) {
        v = create_variable(type,name);
        v->parameter = param;
    } else {
```

```
        if(v->type != type){
            yyerror("Function parameters differ in type.");
        }
    }
    return v;
}

int
print_if (symtabEntry* check)
{
    sprintf(instructions[instructionCounter],"if (%s = 0) goto",check->name);
    return instructionCounter++;
}

int
print_goto ()
{
    sprintf(instructions[instructionCounter],"goto");
    return instructionCounter++;
}

void
print_full_not_if (symtabEntry* check, int target)
{
    sprintf(instructions[instructionCounter],"if (%s = 0) goto %d",check->name,
        instructionCounter+2);
    ++instructionCounter;
    sprintf(instructions[instructionCounter++],"goto %d",target);
}

void
print_full_goto (int target)
{
    sprintf(instructions[instructionCounter++],"goto %d",target);
}

symtabEntry*
print_binary_expression (char* op, symtabEntryType type,
                         symtabEntry* a, symtabEntry* b)
{
    symtabEntry* c = create_helper_variable(type);
    sprintf(instructions[instructionCounter++],"%s := %s %s %s",c->name,a->name,op,b->
        name);

    return c;
}

symtabEntry*
print_unary_expression (char* op, symtabEntryType type, symtabEntry* a)
{
    symtabEntry* c = create_helper_variable(type);
    sprintf(instructions[instructionCounter++],"%s := %s %s",c->name,op,a->name);
    return c;
}

symtabEntry* print_cast(symtabEntry* a, symtabEntryType castTo) {
    if(a->type != castTo){
        a = print_unary_expression(castTo==REAL?"tofloat":"toint",castTo,a);
    }
    return a;
}

symtabEntry*
print_binary_cast_expression ( char* op, symtabEntry* a, symtabEntry* b)
{
    symtabEntryType type = (a->type == INTEGER && b->type == INTEGER)?INTEGER:REAL;

    // cast
    if(type == REAL) {
```

```c
        a = print_cast(a, REAL);
        b = print_cast(b, REAL);
    }

    return print_binary_expression(op,type,a,b);
}

symtabEntry*
print_binary_integer_only_expression (char* op, symtabEntry* a, symtabEntry* b)
{
    if(a->type != INTEGER || a->type != INTEGER){
        yyerror("Passing non integer to interger only operation");
    }
    return print_binary_expression(op,INTEGER,a,b);
}

symtabEntry* print_constant_assignment(symtabEntryType type, char* value) {
    symtabEntry* c = create_helper_variable(type);
    sprintf(instructions[instructionCounter++],"%s := %s",c->name,value);
    return c;
}

symtabEntry* print_variable_assignment(symtabEntry* a,symtabEntry* b) {
    sprintf(instructions[instructionCounter++],"%s := %s",a->name,b->name);
    return a;
}

symtabEntry*
print_left_shift(symtabEntry* a,symtabEntry* b){
    symtabEntry* c = create_helper_variable(INTEGER);
    symtabEntry* r = create_helper_variable(INTEGER);
    print_variable_assignment(c,b);
    print_variable_assignment(r,a);
    sprintf(instructions[instructionCounter],"if (%s <= 0) goto %d",c->name,
        instructionCounter+4);
    sprintf(instructions[instructionCounter+1],"%s := %s * 2",r->name,r->name);
    sprintf(instructions[instructionCounter+2],"%s := %s - 1",c->name,c->name);
    sprintf(instructions[instructionCounter+3],"goto %d",instructionCounter);
    instructionCounter += 4;
    return r;
}

symtabEntry*
print_logical_if(char* op,symtabEntry* a,symtabEntry* b){
    symtabEntry* c = create_helper_variable(INTEGER);
    sprintf(instructions[instructionCounter],"if (%s %s %s) goto %d",a->name, op, b->name
        , instructionCounter+3);
    sprintf(instructions[instructionCounter+1],"%s := %s",c->name,"0");
    sprintf(instructions[instructionCounter+2],"goto %d",instructionCounter+4);
    sprintf(instructions[instructionCounter+3],"%s := %s",c->name,"1");
    instructionCounter += 4;
    return c;
}

void print_pass_param(symtabEntry* a) {
    sprintf(instructions[instructionCounter++],"param %s",a->name);
}

symtabEntry* print_function_call(symtabEntry* f, int params){
    if(f->internType == NOP) {
        sprintf(instructions[instructionCounter++],"call %s, %d",f->name,params);
        return NULL;
    } else {
        symtabEntry* r = create_helper_variable(f->internType);
        sprintf(instructions[instructionCounter++],"%s := call %s, %d",r->name,f->name,
            params);
        return r;
    }
}
```

```

void print_conditional_jump(symtabEntry* boolean) {
    sprintf(instructions[instructionCounter++],"if (%s = 0) goto M",boolean);
}

void print_return(symtabEntry* a) {
    if(a == NULL) {      // void
        if(current_function->internType != NOP) {
            yyerror("Returning nothing from non-void function.");
        }
        sprintf(instructions[instructionCounter++],"return");
    } else {     // non void
        if(current_function->internType == NOP) {
            yyerror("Void function may not return a value.");
        }
        a = print_cast(a, current_function->internType);

        sprintf(instructions[instructionCounter++],"return %s",a->name);
    }
}

%}

// TODO how does a functioncall work? how do i know where to jump?
// TODO how does receiving function parameters work?
// TODO how do jumps work?

// left means, build the tree from left to right

// right left
// bottom high prio
// top low prio

// left assoziativ
// a+b+c = (a+b)+c

// ++a

//Bison declarations

%token CONSTANT
%token DO
%token ELSE
%token FLOAT
%token IDENTIFIER
%token IF
%token INT
%token RETURN
%token VOID
%token WHILE

%token '{'
%token '}'
%token ';'

%token '('
%token ')'
%token ','

%right '='

%left  LOG_AND
%left  LOG_OR
%left  SHIFTLEFT

%left  NOT_EQUAL
%left  EQUAL
%left  GREATER_OR_EQUAL
%left  LESS_OR_EQUAL
```

```
%left   '<'
%left   '>'

%left   '+'
%left   '-'

%left   '*'
%left   '/'
%left   '%'

%left DEC_OP
%left INC_OP
%left '!'
%left U_MINUS
%left U_PLUS

%type <type> INT
%type <type> FLOAT
%type <type> VOID
%type <type> var_type

%type <str> id
%type <type> declaration
%type <entry> expression
%type <str> CONSTANT
%type <entry> assignment
%type <integer> exp_list
%type <integer> parameter_list
%type <integer> if_start
%type <integer> else_start
%type <integer> while_start
%type <integer> do_start

%union
{   // defines yylval
    char str[1000];
    int integer;
    float real;
    symtabEntryType type;
    symtabEntry* entry;
//     struct CharQueue *queue;
}

%%    // grammar rules

programm
    : function
    | programm function
    ;

function_start
    : var_type id
    {
        current_function = declare_function($2,$1);
    }
    '(' parameter_list ')'
    {
        if(current_function->parameter == -1){
            // first declaration
            // update parameter amount in symboltable
            current_function->parameter = $5;
        } else {
            if(current_function->parameter != $5){
                yyerror("Function declared again with wrong amount of parameters.");
            }
        }
    }
    ;

```

```
function
    : function_start ';'
    | function_start
    {
        current_function->line = instructionCounter;
        // reset relative stack pointer
        // assumption: Only ints are passed to function
        rel_addr = current_function->parameter*4;
    }
    function_body
    {
        if(current_function->internType == NOP){
            print_return(NULL);
        }
        current_function->offset = rel_addr;
    }
    ;

function_body
    : '{' statement_list '}'
    | '{' declaration_list statement_list '}'
    ;

declaration_list
    : declaration ';'
    | declaration_list declaration ';'
    ;

declaration
    : INT id
    {
        create_variable(INTEGER, $2);
        $$ = INTEGER;
    }
    | FLOAT id
    {
        create_variable(REAL, $2);
        $$ = REAL;
    }
    | declaration ',' id
    {
        create_variable($1, $3);
        $$ = $1;
    }
    ;

parameter_list
    : INT id
    {
        $$ = 1;
        create_parameter(INTEGER, $2, $$);
    }
    | FLOAT id
    {
        $$ = 1;
        create_parameter(REAL, $2, $$);
    }
    | parameter_list ',' INT id
    {
        $$ = $1+1;
        create_parameter(INTEGER, $4, $$);
    }
    | parameter_list ',' FLOAT id
    {
        $$ = $1+1;
        create_parameter(REAL, $4, $$);
    }
    | VOID
    { $$ = 0; }
```

```
    |
    { $$ = 0; }
    ;

var_type
    : INT    {$$ = INTEGER;}
    | VOID   {$$ = NOP;}
    | FLOAT {$$ = REAL;}
    ;

statement_list
    : statement
    | statement_list statement
    ;

statement
    : matched_statement
    | unmatched_statement
    ;

if_start
    : IF '(' assignment ')'
    {
        $$ = print_if($3);
    }
    ;

else_start
    : if_start matched_statement ELSE
    {
        $$ = print_goto();
        // backpatch if
        patch($1);
    }
    ;

while_start
    : WHILE '(' assignment ')'
    {
        $$ = print_if($3);
    }
    ;

do_start
    : DO
    {
        $$ = instructionCounter;
    }
    ;

matched_statement
    : else_start matched_statement
    {
        // backpatch else
        patch($1);
    }

    | assignment ';'
    | RETURN ';'
    {
        print_return(NULL);
    }
    | RETURN assignment ';'
    {
        print_return($2);
    }
    | while_start matched_statement
    {
        print_full_goto($1);
```

```
        // backpatch while
        patch($1);
    }
    | do_start statement WHILE '(' assignment ')' ';'
    {
        print_full_not_if($5,$1);
    }
    | '{' statement_list '}'
    | '{''}'
    ;

unmatched_statement
    : if_start statement
    {
        // backpatch if
        patch($1);
    }
    | else_start unmatched_statement
    {
        // backpatch else
        patch($1);
    }
    | while_start unmatched_statement
    {
        print_full_goto($1);
        // backpatch while
        patch($1);
    }
    ;

assignment
    : expression
    | id '=' expression
    {
        $$ = variable_lookup($1);
        if($$ == NULL){
            yyerror("Assignment to undeclared variable.");
        }
        print_variable_assignment($$,$3);
    }
    ;

expression
    : INC_OP expression
    {
        symtabEntry* c = print_constant_assignment($2->type, "1");
        $$ = print_binary_expression("+",$2->type,$2,c);
    }
    | DEC_OP expression
    {
        symtabEntry* c = print_constant_assignment($2->type, "1");
        $$ = print_binary_expression("-",$2->type,$2,c);
    }
    | expression LOG_OR expression
    {
        $$ = print_binary_integer_only_expression("+",$1,$3);
    }
    | expression LOG_AND expression
    {
        $$ = print_binary_integer_only_expression("*",$1,$3);
    }
    | expression NOT_EQUAL expression
    {
        $$ = print_logical_if("!=",$1,$3);
    }
    | expression EQUAL expression
    {
        $$ = print_logical_if("=",$1,$3);
    }
```

```
        | expression GREATER_OR_EQUAL expression
        {
           $$ = print_logical_if(">=",$1,$3);
        }
        | expression LESS_OR_EQUAL expression
        {
           $$ = print_logical_if("<=",$1,$3);
        }
        | expression '>' expression
        {
           $$ = print_logical_if(">",$1,$3);
        }
        | expression '<' expression
        {
           $$ = print_logical_if("<",$1,$3);
        }
        | expression SHIFTLEFT expression
        {
           $$ = print_left_shift($1,$3);
        }
        | expression '+' expression
        {
           $$ = print_binary_cast_expression("+",$1,$3);
        }
        | expression '-' expression
        {
           $$ = print_binary_cast_expression("-",$1,$3);
        }
        | expression '*' expression
        {
           $$ = print_binary_cast_expression("*",$1,$3);
        }
        | expression '/' expression
        {
           $$ = print_binary_cast_expression("/",$1,$3);
        }
        | expression '%' expression
        {
           $$ = print_binary_integer_only_expression("%",$1,$3);
        }
        | '!' expression
        {
           symtabEntry* c = print_constant_assignment($2->type, "1");
           $$ = print_binary_expression("-",$2->type,c,$2);
        }

        | '+' expression %prec U_PLUS
        {
           $$ = $2;
        }
        | '-' expression %prec U_MINUS
        {
           $$ = print_unary_expression("-",INTEGER,$2);
        }

        | CONSTANT
        {
           $$ = print_constant_assignment(strchr($1,'.')==NULL ? INTEGER:REAL, $1);
        }
        | '(' expression ')'
        {
           $$ = $2;
        }
        | id '(' exp_list ')' {
           symtabEntry* e = lookup($1);
           if(e == NULL) {
               yyerror("Trying to call undecared function");
           }
           $$ = print_function_call(e,$3);
```

```
    }
    | id
    {
        $$ = variable_lookup($1);
        if($$ == NULL){
            yyerror("Usage of undeclared variable.");
        }
    }
    ;

exp_list
    : {$$ = 0;}
    | expression    {print_pass_param($1); $$ = 1;}
    | exp_list ',' expression   {print_pass_param($3); $$ = ++$1;}
    ;

id
    : IDENTIFIER  { strcpy($$,yylval.str); }
    ;

%%

int main() {
    declare_function("main", INTEGER);
    yyparse();

    int i;

    writeSymboltable(theSymboltable, stdout);

    printf("\nCODE\n------------------------------------\n");


    for(i=0 ; i<instructionCounter ; ++i){
        printf("%d\t%s\n",i,instructions[i]);
    }

    return 0;
}

void yyerror(char * message) {
    printf("error message <%s>\n",message);
    exit(1);
}

//Epilogue
```

2.5 Explanation

Given lex.y file is code for our lexical analyser which is takes a C program as input and scan and analyses that code and works as given below.

2.5.1 Alphabet keywords

They are already predefined in their respective section.

2.5.2 To keep track of code blocks

This will take a section which is between/before or after comments (excludes comments).

2.5.3 To resolve nested comments

We resolved this situtation by keeping integer track of the depth of comments.

2.5.4 String errors

These are solved by checking last character of given string to check if it has ended or not.

3 Test Cases

3.1 Semantic Analysis - Program without errors

```
int main()
{
    char m, y, z, w;
    int b = 2 * 3 - 10 ;
    int x = y + (z*w) / 2+2*m-2;
    if( 10 + 15 < 12 * 2 -10 /2 )
    {
        int a = 10;
        if( a < 890 ){
            int c = 56789;
        }
        else
        {
            int j = 12412;
        }
    }
    else{

        int my_val = 1234234 ;
    }
    while(a < 10){
        int d = 100;
        if(a < 100){
            int e= 11247098;
        }
        else {int r = 10000;}
    }

    main(b);
}
int my_func(int f) {
    return a+2;
}
```

3.2 Semantic Analysis - Program with errors

```
int main()
{
    char m, y, z, w;
    int b = 2 * 3 - 10 ;
    int x = y + (z*w) / 2+2*m-2;
    int b = (x+m)*(z*w)*64;
    if( 10 + 15 < 12 * 2 -10 /2 )
    {
        int a = 10;
        if( a < 890 ){
            int c = 56789;
        }
        else
        {
```

```
            int j = 12412;
        }
    }
    else{

        int my_val = 1234234 ;
    }
    while(a < 10){
        int d = 100;
        if(a < 100){
            int b = 11247098;
        }
        else {int a = 10000;}
    }

    my_func(b);
}
int my_func(int a) {
    return a+2;
}
```

3.3 Code Generation - Input Program

```
float questionmark (int x, int y);

int main (void){
    int variable_1;
    int variable_2, variable_4;
    float variable_3, variable_5;

    variable_1 = 1;
    variable_2 = 2;

    if(! variable_3){
        return 0;
    }else{
        return 1;
    }
}

float questionmark (int x, int y){
    float result;


    return result;
}
```

3.4 Code Generation - Input Program with Functions

```
float questionmark (int x, int y);

int main (void){

    int variable_1;
    int variable_2;
    float variable_3;

    variable_1 = 1;
    variable_2 = 2;

    while (variable_1 <= 10 && 1){

        variable_3 = questionmark (variable_1, variable_2);

        variable_2 = ++variable_1;
        variable_2 = variable_2 << 2;
    }
```

```
    if(! variable_3){
        return 0;
    }else{
        return 1;
    }
}

float questionmark (int x, int y){

    float result;

    do{
        if (y/x){
            result = 1.0;
        }
        --y;
    }while( y < x || 0);

    return result;
}
```

4 Implementation

For implementation of this given code we used following technique:

4.1 Checking for general expressions

```
primary_expression
: IDENTIFIER {printf("primary_expression -> IDENTIFIER\n");}
| CONSTANT {printf("primary_expression -> CONSTANT\n");}
| STRING_LITERAL {printf("primary_expression -> STRING_LITERAL\n");}
| '(' expression ')'{printf("primary_expression -> ( expression )\n");}
;

postfix_expression
: primary_expression {printf("postfix_expression -> primary_expression\n");}
| postfix_expression '[' expression ']'{printf("postfix_expression -> postfix_expression [ expressio
| postfix_expression '(' ')'{printf("postfix_expression -> postfix_expression ( )\n");}
| postfix_expression '(' argument_expression_list ')'{printf("postfix_expression -> postfix_expressi
| postfix_expression '.' IDENTIFIER {printf("postfix_expression -> postfix_expression . IDENTIFIER\n
| postfix_expression PTR_OP IDENTIFIER {printf("postfix_expression -> postfix_expression PTR_OP IDEN
| postfix_expression INC_OP {printf("postfix_expression -> postfix_expression INC_OP\n");}
| postfix_expression DEC_OP {printf("postfix_expression -> postfix_expression DEC_OP\n");}
;

argument_expression_list
: assignment_expression {printf("argument_expression_list -> assignment_expression\n");}
| argument_expression_list ',' assignment_expression {printf("argument_expression_list -> argument_e
;

unary_expression
: postfix_expression {printf("unary_expression -> postfix_expression\n");}
```

```
| INC_OP unary_expression {printf("unary_expression -> INC_OP unary_expression\n");}
| DEC_OP unary_expression {printf("unary_expression -> DEC_OP unary_expression\n");}
| unary_operator cast_expression {printf("unary_expression -> unary_operator cast_expression\n");}
| SIZEOF unary_expression {printf("unary_expression -> SIZEOF unary_expression\n");}
| SIZEOF '(' type_name ')'{printf("unary_expression -> SIZEOF ( type_name )\n");}
;
```

4.2 Checking for numerical expressions

```
cast_expression
: unary_expression {printf("cast_expression -> unary_expression\n");}
| '(' type_name ')' cast_expression {printf("cast_expression -> ( type_name ) cast_expression\n");}
;

multiplicative_expression
: cast_expression {printf("multiplicative_expression -> cast_expression\n");}
| multiplicative_expression '*' cast_expression {printf("multiplicative_expression -> multiplicative
| multiplicative_expression '/' cast_expression {printf("multiplicative_expression -> multiplicative
| multiplicative_expression '%' cast_expression {printf("multiplicative_expression -> multiplicative
;

additive_expression
: multiplicative_expression {printf("additive_expression -> multiplicative_expression\n");}
| additive_expression '+' multiplicative_expression {printf("additive_expression -> additive_express
| additive_expression '-' multiplicative_expression {printf("additive_expression -> additive_express
;

shift_expression
: additive_expression {printf("shift_expression -> additive_expression\n");}
| shift_expression LEFT_OP additive_expression {printf("shift_expression -> shift_expression LEFT_OP
| shift_expression RIGHT_OP additive_expression {printf("shift_expression -> shift_expression RIGHT_
;
```

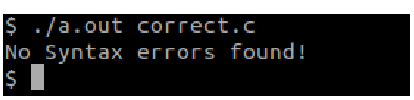
4.3 Checking for relational and logical expressions

```
relational_expression
: shift_expression {printf("relational_expression -> shift_expression\n");}
| relational_expression '<' shift_expression {printf("relational_expression -> relational_expression
| relational_expression '>' shift_expression {printf("relational_expression -> relational_expression
| relational_expression LE_OP shift_expression {printf("relational_expression -> relational_expressi
| relational_expression GE_OP shift_expression {printf("relational_expression -> relational_expressi
;

equality_expression
```

```
: relational_expression {printf("equality_expression -> relational_expression\n");}
| equality_expression EQ_OP relational_expression {printf("equality_expression -> equality_expressio
| equality_expression NE_OP relational_expression {printf("equality_expression -> equality_expressio
;

and_expression
: equality_expression {printf("and_expression -> equality_expression\n");}
| and_expression ’&’ equality_expression {printf("and_expression -> and_expression & equality_expres
;

exclusive_or_expression
: and_expression {printf("exclusive_or_expression -> and_expression\n");}
| exclusive_or_expression ’^’ and_expression {printf("exclusive_or_expression -> exclusive_or_expres
;

inclusive_or_expression
: exclusive_or_expression {printf("inclusive_or_expression -> exclusive_or_expression\n");}
| inclusive_or_expression ’|’ exclusive_or_expression {printf("inclusive_or_expression -> inclusive_
;

logical_and_expression
: inclusive_or_expression {printf("logical_and_expression -> inclusive_or_expression\n");}
| logical_and_expression AND_OP inclusive_or_expression {printf("logical_and_expression -> logical_a
;

logical_or_expression
: logical_and_expression {printf("logical_or_expression -> logical_and_expression\n");}
| logical_or_expression OR_OP logical_and_expression {printf("logical_or_expression -> logical_or_ex
;

conditional_expression
: logical_or_expression {printf("conditional_expression -> logical_or_expression\n");}
| logical_or_expression ’?’ expression ’:’ conditional_expression {printf("conditional_expression ->
;
```

5 Results

5.1 Semantic Analysis Successful Run Part 1



The program table is as follows:

```
Type    Symbol
3       f
3       my_func
3       r
3       e
3       d
3       my_val
3       j
3       c
3       a
3       x
3       b
2       w
2       z
2       y
2       m
3       main
```

5.2 Semantic Analysis Successful Run Part 2

```
$ lex scan.l
$ yacc parse.y
$ gcc lex.yy.c y.tab.c -w
$ ./a.out incorrect.c
 VOID=1  CHAR=2  INT=3  FLOAT=4  DOUBLE
 ***********************************************************************************************

A Semantic error has been encountered at Pos : 6 : 7 : b is already defined as 3

************************************************************************************************
 VOID=1  CHAR=2  INT=3  FLOAT=4  DOUBLE
 ***********************************************************************************************

A Semantic error has been encountered at Pos : 25 : 9 : b is already defined as 3

************************************************************************************************
 VOID=1  CHAR=2  INT=3  FLOAT=4  DOUBLE
 ***********************************************************************************************

A Semantic error has been encountered at Pos : 27 : 14 : a is already defined as 3

************************************************************************************************
************************************************************************************************

A Semantic error has been encountered at Pos : 30 : 10 : my_func is an undeclared identifier

************************************************************************************************
 VOID=1  CHAR=2  INT=3  FLOAT=4  DOUBLE
 ***********************************************************************************************

A Semantic error has been encountered at Pos : 32 : 18 : a is already defined as 3

************************************************************************************************
No Syntax errors found!
$
```

The program table is as follows:

```
Type     Symbol
3    my_func
3    d
3    my_val
3    j
3    c
3    a
3    x
3    b
2    w
2    z
2    y
2    m
3    main
```

5.3 Code Generation - Simple

```
SYMBOLS
------------------------------------------
Name                          Type   Int_Typ  Offset  Line   Index1  Index2  Parent   Parameter
---------------------------------------------------------------------------------------------------

main                          Main   Int      60      0      0       0       None         0
questionmark                  Func   Real     16      12     0       0       None         2
x                             Int    None     0       0      0       0       questionmark
          1
y                             Int    None     4       0      0       0       questionmark
          2
variable_1                    Int    None     0       0      0       0       main         0
variable_2                    Int    None     4       0      0       0       main         0
variable_4                    Int    None     8       0      0       0       main         0
variable_3                    Real   None     12      0      0       0       main         0
variable_5                    Real   None     20      0      0       0       main         0
V__H0                         Int    None     28      0      0       0       main         0
V__H1                         Int    None     32      0      0       0       main         0
V__H2                         Real   None     36      0      0       0       main         0
V__H3                         Real   None     44      0      0       0       main         0
V__H4                         Int    None     52      0      0       0       main         0
V__H5                         Int    None     56      0      0       0       main         0
result                        Real   None     8       0      0       0       questionmark
          0

CODE
------------------------------------------
0    V__H0 := 1
1    variable_1 := V__H0
2    V__H1 := 2
3    variable_2 := V__H1
4    V__H2 := 1
5    V__H3 := V__H2 - variable_3
6    if (V__H3 = 0) goto 10
7    V__H4 := 0
8    return V__H4
9    goto 12
10   V__H5 := 1
11   return V__H5
12   return result
```

5.4 Code Generation - Simple with Functions

```
SYMBOLS
------------------------------------------
Name                          Type   Int_Typ  Offset  Line   Index1  Index2  Parent   Parameter
```

```
------------------------------------------------------------------------------------------------

main                     Main    Int     92      0       0       0       None          0
questionmark             Func    Real    48      36      0       0       None          2
x                        Int     None    0       0       0       0       questionmark
        1
y                        Int     None    4       0       0       0       questionmark
        2
variable_1               Int     None    0       0       0       0       main          0
variable_2               Int     None    4       0       0       0       main          0
variable_3               Real    None    8       0       0       0       main          0
V__H0                    Int     None    16      0       0       0       main          0
V__H1                    Int     None    20      0       0       0       main          0
V__H2                    Int     None    24      0       0       0       main          0
V__H3                    Int     None    28      0       0       0       main          0
V__H4                    Int     None    32      0       0       0       main          0
V__H5                    Int     None    36      0       0       0       main          0
V__H6                    Real    None    40      0       0       0       main          0
V__H7                    Int     None    48      0       0       0       main          0
V__H8                    Int     None    52      0       0       0       main          0
V__H9                    Int     None    56      0       0       0       main          0
V__H10                   Int     None    60      0       0       0       main          0
V__H11                   Int     None    64      0       0       0       main          0
V__H12                   Real    None    68      0       0       0       main          0
V__H13                   Real    None    76      0       0       0       main          0
V__H14                   Int     None    84      0       0       0       main          0
V__H15                   Int     None    88      0       0       0       main          0
result                   Real    None    8       0       0       0       questionmark
        0
V__H16                   Int     None    16      0       0       0       questionmark
        0
V__H17                   Real    None    20      0       0       0       questionmark
        0
V__H18                   Int     None    28      0       0       0       questionmark
        0
V__H19                   Int     None    32      0       0       0       questionmark
        0
V__H20                   Int     None    36      0       0       0       questionmark
        0
V__H21                   Int     None    40      0       0       0       questionmark
        0
V__H22                   Int     None    44      0       0       0       questionmark
        0

CODE
-----------------------------------
0    V__H0 := 1
1    variable_1 := V__H0
2    V__H1 := 2
3    variable_2 := V__H1
4    V__H2 := 10
5    if (variable_1 <= V__H2) goto 8
6    V__H3 := 0
7    goto 9
8    V__H3 := 1
9    V__H4 := 1
10   V__H5 := V__H3 * V__H4
11   if (V__H5 = 0) goto 28
12   param variable_1
13   param variable_2
14   V__H6 := call questionmark, 2
15   variable_3 := V__H6
16   V__H7 := 1
17   V__H8 := variable_1 + V__H7
18   variable_2 := V__H8
19   V__H9 := 2
20   V__H10 := V__H9
21   V__H11 := variable_2
22   if (V__H10 <= 0) goto 26
```

```
23  V__H11 := V__H11 * 2
24  V__H10 := V__H10 - 1
25  goto 22
26  variable_2 := V__H11
27  goto 11
28  V__H12 := 1
29  V__H13 := V__H12 - variable_3
30  if (V__H13 = 0) goto 34
31  V__H14 := 0
32  return V__H14
33  goto 36
34  V__H15 := 1
35  return V__H15
36  V__H16 := y / x
37  if (V__H16 = 0) goto 40
38  V__H17 := 1.0
39  result := V__H17
40  V__H18 := 1
41  V__H19 := y - V__H18
42  if (y < x) goto 45
43  V__H20 := 0
44  goto 46
45  V__H20 := 1
46  V__H21 := 0
47  V__H22 := V__H20 + V__H21
48  if (V__H22 = 0) goto 50
49  goto 36
50  return result
```

References

[1] En.wikibooks.org, (2016). Compiler Construction/Lexical analysis - Wikibooks, open books for an open world. [online] Available at: https://en.wikibooks.org/wiki/Compiler _Construction/Syntax _analysis

[2] Cs.fsu.edu, (2016). [online] Available at: http://www.cs.fsu.edu/ xyuan/-cop5621/lect3_syntax.ppt

[3] dragonbook.stanford.edu/lecture-notes/05-Syntax-Analysis.pdf

[4] cecs.wright.edu/ tkprasad/courses/cs780/L3Lexing.pdf

[5] quex.sourceforge.net/

[6] http://www.pling.org.uk/cs/lsa.html

[7] www.cs.nyu.edu/courses/spring11/G22.2130-001/lecture7.pdf