**COMPILER DESIGN: PHASE 1 REPORT**

A **compiler** is [computer software](https://en.wikipedia.org/wiki/Computer_software) that transforms computer code written in one [programming language](https://en.wikipedia.org/wiki/Programming_language) (the source language) into another programming language (the target language). Compilers are a type of [translator](https://en.wikipedia.org/wiki/Translator_(computing)) that support digital devices, primarily computers. The name *compiler* is primarily used for programs that translate [source code](https://en.wikipedia.org/wiki/Source_code) from a [high-level programming language](https://en.wikipedia.org/wiki/High-level_programming_language) to a [lower level language](https://en.wikipedia.org/wiki/Lower_level_language) (e.g., [assembly language](https://en.wikipedia.org/wiki/Assembly_language), [object code](https://en.wikipedia.org/wiki/Object_code), or [machine code](https://en.wikipedia.org/wiki/Machine_code)) to create an [executable](https://en.wikipedia.org/wiki/Executable) program.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

We basically have two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. Synthesis phase creates an equivalent target program from the intermediate representation.

The compiler has two modules namely front end and back end. Front-end constitutes of the Lexical analyzer, semantic analyzer, syntax analyzer and intermediate code generator. And the rest are assembled to form the back end.

1. **Lexical Analyzer –** It reads the program and converts it into tokens. It converts a stream of lexemes into a stream of tokens. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes white-spaces and comments.
2. **Syntax Analyzer –** It is sometimes called as parser. It constructs the parse tree. It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.  
   The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.

Syntax error can be detected at this level if the input is not in accordance with the grammar.

1. **Semantic Analyzer –** It verifies the parse tree, whether it’s meaningful or not. It furthermore produces a verified parse tree.
2. **Intermediate Code Generator –** It generates intermediate code that is a form which can be readily executed by machine. We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

1. **Code Optimizer –** It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimization can be categorized into two types: machine dependent and machine independent.
2. **Target Code Generator –** The main purpose of Target Code generator is to write a code that the machine can understand. The output is dependent on the type of assembler. This is the final stage of compilation.

Common terminologies encountered

**Symbol Table –** It is a data structure being used and maintained by the compiler, consists all the identifier’s name along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

Lexical Analysis

**Lexical analysis** or **tokenization** is the process of converting a sequence of characters into a sequence of tokens (strings with an assigned and thus identified meaning). A program that performs lexical analysis may be termed a *lexer*, *tokenizer,* or *scanner*. A lexer is generally combined with a parser, which together analyze the syntax of programming languages.

**Lexeme –** A *lexeme* is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

**Token –** A *lexical token* or simply *token* is a pair consisting of a *token name* and an optional *token value*. The token name is a category of lexical unit. Common token names are

* Identifiers: names the programmer chooses;
* Keywords: names already in the programming language;
* Separators (also known as punctuators): punctuation characters and paired-delimiters;
* Operators: symbols that operate on arguments and produce results;
* Literals: numeric, logical, textual, reference literals;
* Comments: line, block.

All tokens recognized are then stored in the **symbol table** except the literals. The literals are stored separately in **constant table**. Constant tables is similar to symbol table in terms of its structure and use.

Identifying looping contraints(for and while), if contraints, functions, arrays and structures will be handled in phase 2 of compiler design, i.e. by syntax analyzer (parser).

Code

%{

#include<stdio.h>

#include<string.h>

struct hashtable{

char name[105];

char type[105];

int len;

}table[1005];

struct consttable{

char name[105];

char type[105];

int len;

}ctable[1005];

int Hash(char \*s){

int mod=1001;

int l=strlen(s),val=0,i;

for (i=0;i<l;i++){

val=val\*10+(s[i]-'A');

val=val%mod;

while(val<0){

val+=mod;

}

}

return val;

}

void insert(char \*arg1,char \*arg2){

int l1=strlen(arg1);

int l2=strlen(arg2);

int v=Hash(arg1);

if(table[v].len==0){

strcpy(table[v].name,arg1);

strcpy(table[v].type,arg2);

table[v].len=strlen(arg1);

return ;

}

if(strcmp(table[v].name,arg1)==0)

return ;

int i,pos=0;

for (i=0;i<1001;i++){

if(table[i].len==0){

pos=i;

break;

}

}

strcpy(table[pos].name,arg1);

strcpy(table[pos].type,arg2);

table[pos].len=strlen(arg1);

}

void insert2(char \*arg1,char \*arg2){

int l1=strlen(arg1);

int l2=strlen(arg2);

int v=Hash(arg1);

if(ctable[v].len==0){

strcpy(ctable[v].name,arg1);

strcpy(ctable[v].type,arg2);

ctable[v].len=strlen(arg1);

return ;

}

if(strcmp(ctable[v].name,arg1)==0)

return ;

int i,pos=0;

for (i=0;i<1001;i++){

if(ctable[i].len==0){

pos=i;

break;

}

}

strcpy(ctable[pos].name,arg1);

strcpy(ctable[pos].type,arg2);

ctable[pos].len=strlen(arg1);

}

void print(){

int i;

for ( i=0;i<1001;i++){

if(table[i].len==0){

continue;

}

printf("%s \t %s\n",table[i].name,table[i].type);

}

}

void print2(){

int i;

for ( i=0;i<1001;i++){

if(ctable[i].len==0){

continue;

}

printf("%s \t %s\n",ctable[i].name,ctable[i].type);

}

}

%}

LEQ <=

GEQ >=

EQ =

LES <

GRE >

PLUS \+

MINUS \-

MULT \\*

DIV \/

REM %

AND &

OR \|

%%

[ \n\t] ;

\".\*\"|\'.\*\' {printf("%s \t- STRING CONSTANT\n", yytext); insert2(yytext,"STRING CONSTANT");}

; {printf("%s \t- SEMICOL DELIMITER\n", yytext); insert(yytext, "SEMICOL DELIMITER");}

, {printf("%s \t- COMM DELIMITER\n", yytext); insert(yytext, "COMM DELIMITER");}

\{ {printf("%s \t- OPENING BRACES\n", yytext); insert(yytext, "OPENING BRACES");}

\} {printf("%s \t- CLOSING BRACES\n", yytext); insert(yytext, "CLOSING BRACES");}

\( {printf("%s \t- OPENING BRACKETS\n", yytext); insert(yytext, "OPENING BRACKETS");}

\) {printf("%s \t- CLOSING BRACKETS\n", yytext); insert(yytext, "CLOSING BRACKETS");}

# {printf("%s \t- PREPROCESSOR\n", yytext); insert(yytext, "PREPROCESSOR");}

printf {printf("%s \t- PRINT\n", yytext); insert(yytext, "PRINT");}

\" {printf("%s \t- DQOUTE\n", yytext); insert(yytext, "DQOUTE");}

\' {printf("%s \t- SQOUTE\n", yytext); insert(yytext, "SQOUTE");}

\\ {printf("%s \t- FSLASH\n", yytext); insert(yytext, "FSLASH");}

\. {printf("%s \t- DOT DELIMITER\n", yytext); insert(yytext, "DOT DELIMITER");}

\/\/.\* {printf("%s \t- SINGLE LINE COMMENT\n", yytext); insert(yytext, "SINGLE LINE COMMENT");}

"/\*"([^\*]|\\*+[^\*/])\*\\*+"/" {printf("%s \t- MULTI LINE COMMENT\n", yytext); insert(yytext, "MULTI LINE COMMENT");}

auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|while|main {printf("%s \t- KEYWORD\n", yytext); insert(yytext, "KEYBOARD");}

[a-z|A-Z]([a-z|A-Z]|[0-9])\* {printf("%s \t- IDENTIFIER\n", yytext); insert(yytext, "IDENTIFIER");}

[1-9][0-9]\*|0 {printf("%s \t- NUMBER CONSTANT\n", yytext); insert2(yytext, "NUMBER CONSTANT");}

{PLUS}|{MINUS}|{MULT}|{DIV}|{EQ}|{LEQ}|{GEQ}|{LES}|{GRE}|{REM}|{AND}|{OR} {printf("%s \t- OPERATOR\n", yytext); insert(yytext, "OPERATOR");}

(.?) {printf("%s \t- INVALID\n", yytext); insert(yytext, "INVALID");}

%%

int main(){

int i;

for (i=0;i<1001;i++){

table[i].len=0;

}

yyin=fopen("test-1.c","r");

yylex();

printf("\n\nSYMBOL TABLE\n\n");

print();

printf("\n\nCONSTANT TABLE\n\n");

print2();

}

int yywrap(){

return 1;

}

Screenshots





