



**SAN JOSÉ STATE  
UNIVERSITY**

A

Design Documentation  
On

**COMPILER DESIGNING**  
Submitted to

Prof. Hungwen Li  
(CMPE 220)

by  
Rimjhim Ghosh (011416002)  
Rajeev Sawant (011418030)  
**(Group 14)**

Department of Computer Engineering

October 26, 2017

# TABLE OF CONTENTS

<b>1. Introduction</b>	<b>2</b>
1.1. Purpose	3
1.2. Scope	3
1.3. Overview	5
1.4. Reference Material	5
1.5. Definitions and Acronyms	7
<b>2. System Overview</b>	
<b>3. System Architecture</b>	<b>9</b>
3.1. Architectural Design	10
3.2. Decomposition Description	12
3.3. Design Rationale	
<b>4. Data Design</b>	<b>13</b>
4.1. Data Description	15
4.2. Data Dictionary	
<b>5. Component Design</b>	<b>17</b>
<b>6. Human Interface Design</b>	
6.1. Overview of User Interface	19
6.2. Screen Images	20
<b>7. Requirements Matrix</b>	
<b>8. Appendices</b>	<b>23</b>
	25

# 1. INTRODUCTION

## 1.1. Purpose

Writing a program in machine language is a tedious and error-prone process thus programmers choose to write code in high-level programming language. This language is very different from the machine language that the computer can execute. This is where compiler comes in.

A compiler translates high-level language to machine language. And during this process it reports mistakes made by programmer.

Using a high-level language for programming has a large impact on how fast programs can be developed. The main reasons for this are:

- Compared to machine language, the notation used by them are closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.

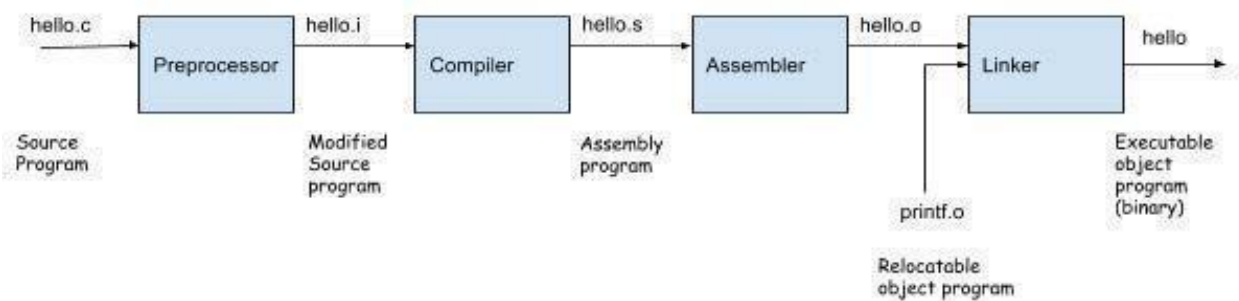
Another advantage of writing a code in high-level language is that when converted to machine language can be brought to run in different platforms.

A compiler's performance is judged by the difference of speed between a hand-written machine code and translating a well-structured program.

## 1.2. Scope

In this project, we would introduce COOL programming language translators and provide the overview of a typical compiler. We will also provide detail of COOL programming languages and machine architecture that are shaping compilers. We would also include an overview on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation.

### 1.3. Overview



**FIG1: Programming language Processing Stages**

#### Preprocessor Phase:

Modifies the original C program according to the directive that begins with '#' character. Preprocessor deals with macro processing, augmentation, file inclusion, language extension etc.

#### Compilation Phase:

The Compiler translates High level language into low level machine language. A compiler reads whole code at once, creates tokens, checks semantics, generates intermediate code and executes the whole program. The compiler reads the whole program and doesn't stop execution even in case of error. In accordance to above diagram the compiler translates the text file hello.i into the text file hello.s, which contains an assembly-language program.

#### Assembly Phase:

An assembler translates assembly language program into machine code. The output of the assembler is called an object file. In this stage, the assembler translates the hello.s into machine language instruction and stores the result in object file hello.o.

#### Linking Phase:

Linker is a computer program that links and merges various object files together to make an executable file. All this file might have been compiled by separate assemblers. In our case the linker combines the hello.o file and printf.o file to generate the executable file hello.

#### Optimization:

Optimization of the compiler code where, the compiler tries to minimize the time taken to execute a program by minimizing or maximizing some attributes of the executable computer program.

## 1.4. Reference Material

- [1] <https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/>
- [2] <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.580>
- [3] <http://dinosaur.compilertools.net/lex/index.html>
- [4] <http://dinosaur.compilertools.net/#yacc>
- [5] <https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/info>

## 1.5. Definitions and Acronyms

Object.copy	A procedure returning a fresh copy of the object passed in \$a0 Result will be in \$a0
Object.abort	A procedure that prints out the class name of the object in \$a0 Terminates program execution
Object.type_name	Returns the name of the class of object passed in \$a0 as a string object Uses the class tag and the table class_nameTab
IO.out_string	The value of the string object on top of the stack is printed to the terminal. Does not modify \$a0.
IO.out_int	The integer value of the Int object on top of the stack is printed to the terminal. Does not modify \$a0.
IO.in_string	Reads a string from the terminal and returns the read string object in \$a0. (The newline that terminates the input is not part of the string)
IO.in_int	Reads an integer from the terminal and returns the read int object in \$a0.
String.length	Returns the integer object which is the length of the string object passed in \$a0. Result in \$a0.
String.concat	Returns a new string, made from concatenating the string object on top of the stack to the string object in \$a0. Return value in \$a0
String.substr	Returns the substring of the string object passed in \$a0, from index i with length l. The length is defined by the integer object on top of the stack, and the index by the integer object on the stack below l. Result in \$a0.

<code>equality_test</code>	Tests whether the objects passed in <code>\$t1</code> and <code>\$t2</code> have the same primitive type {Int,String,Bool} and the same value. If they do, the value in <code>\$a0</code> is returned, otherwise <code>\$a1</code> is returned.
<code>_dispatch_abort</code>	Called when a dispatch is attempted on a void object. Prints the line number, from <code>\$t1</code> , and filename, from <code>\$a0</code> , at which the dispatch occurred, and aborts.
<code>_case_abort</code>	Should be called when a case statement has no match. The class name of the object in <code>\$a0</code> is printed, and execution halts.
<code>_case_abort2</code>	Called when a case is attempted on a void object. Prints the line number, from <code>\$t1</code> , and filename, from <code>\$a0</code> , at which the dispatch occurred, and aborts.

**Table 1: Acronyms defined in the runtime system.**

## 2. SYSTEM OVERVIEW

Compiler designing requires a development environment like LINUX (Bodhi 1.4.0) using VirtualBox VM. The following are the platforms and environments used.

- COOL: It is a small language that can be implemented with reasonable effort. Cool retains many of the features of modern programming languages including objects, static typing, and automatic memory management.
- Syntax Rules: This section formally defines the type rules of Cool. The type rules define the type of every Cool expression in each context. The context is the type environment, which describes the type of every unbound identifier appearing in an expression.
- FLEX: Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). It's used as the lex implementation together with Yacc parser generator. Flex, an automatic lexical analyser, is often used with Bison, to tokenize input data and provide Bison with tokens.
- BISON: GNU bison, commonly known as Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser that



reads sequences of tokens and decides whether the sequence follows the syntax specified by the grammar. Bison by default generates LALR parsers but can also create GLR parsers. It is compatible with yacc, but also has several extensions over this earlier program

- References on MIPS & SPIM: In our project, we use SPIM Simulator. spim is a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for this processor. spim also provides a simple debugger and minimal set of operating system services. spim does not execute binary (compiled) programs.

## Constraints:

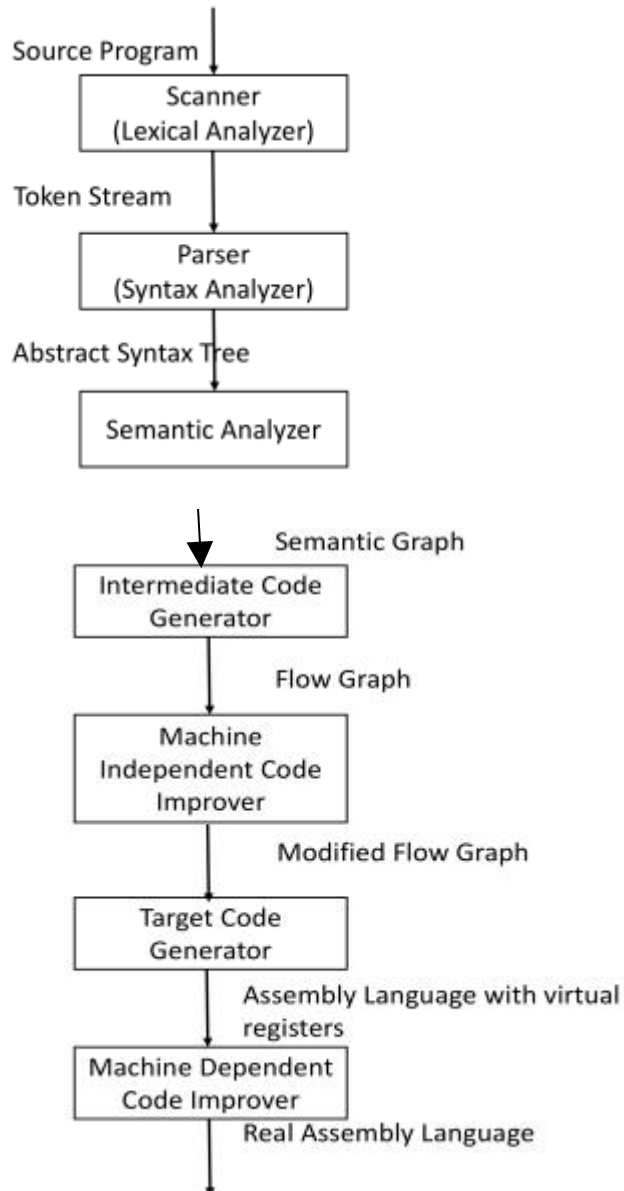
1. Time complexity: Compiler usually have time constraints; the performance of a compiler is judged by its speed. Using flex analyzer has a time complexity of  $O(n)$ . in the length of input. Using extremely long tokens can also cause non-linear performance of scanner.
2. Reentrancy: By default, the scanner used in Flex is not reentrant. This can cause problem for codes that use different threads.
3. Platform specific: Normally the generated scanner contains references to `unistd.h` header file which is Unix specific.
4. Using flex from other languages: Flex can only generate code for C and C++. To use the scanner code generated by flex from other languages a language binding tool such as SWIG can be used.

### 3. System Architecture

#### [3.1] Architecture Design:

The below flow chart shows the main functional components of the compiler to generate assembly code.

We can include an **error recovery system** and **symbol table manager** too to make



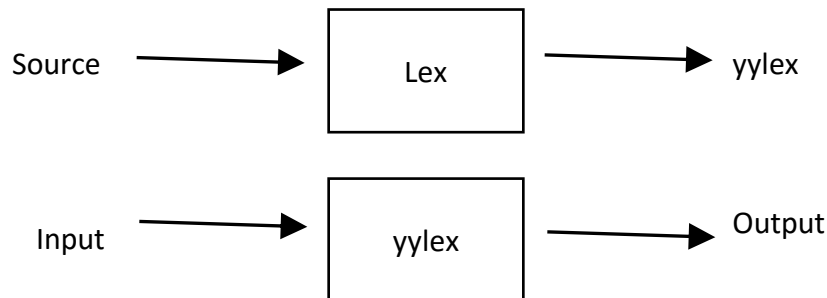
compiler more efficient.

**FIG2: Compiler Architecture**

### [3.2] Decomposition Description:

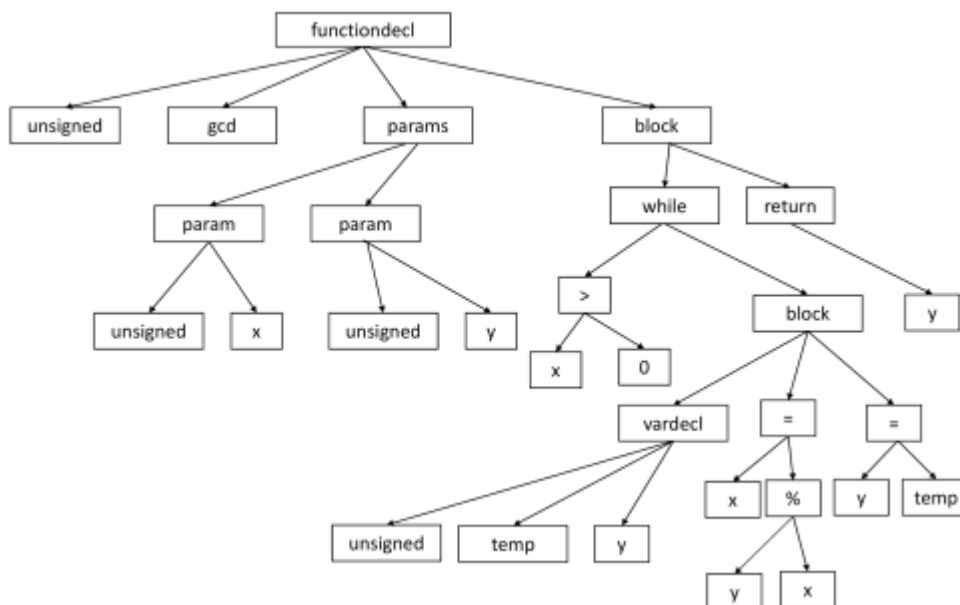
#### Architecture for Lexical Analysis:

Lex turns the user's expressions and actions into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream perform the specified actions for each expression as it is detected



**FIG3: Lexical Analysis Architecture**

#### Architecture for Parsing Analysis:



The parser turns the tokens to abstract syntax tree(AST).

## Architecture for Semantic Analysis:

Semantic analysis provides the meaning of the construction of tokens and syntax structure. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

1.	<i>statements</i>	$\rightarrow$	$\vdash$
2.		$ $	<i>expression ; statements</i>
3.	<i>expression</i>	$\rightarrow$	<i>term expression'</i>
4.	<i>expression'</i>	$\rightarrow$	$+$ <i>term expression'</i>
5.		$ $	$\epsilon$
6.	<i>term</i>	$\rightarrow$	<i>factor term'</i>
7.	<i>term'</i>	$\rightarrow$	$*$ <i>factor term'</i>
8.		$ $	$\epsilon$
9.	<i>factor</i>	$\rightarrow$	<b>number</b>
10.		$ $	$($ <i>expression</i> $)$

Table 2: Semantic rules

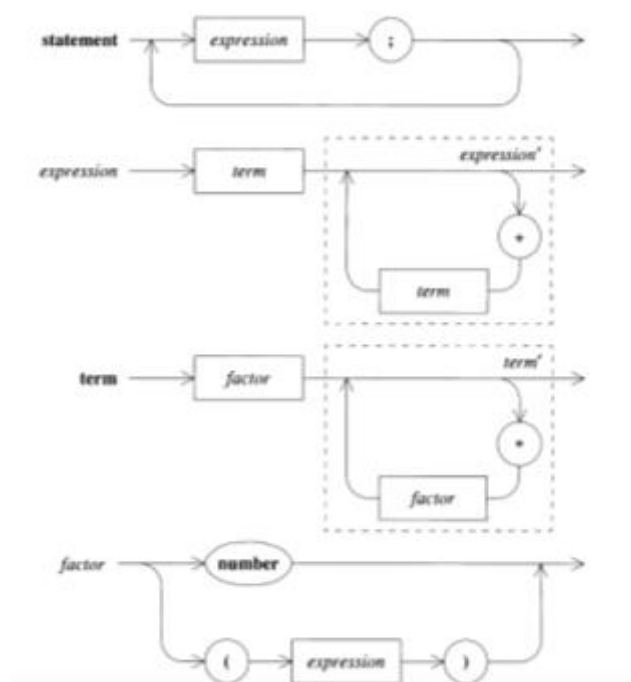


FIG 4: Syntax flow of a regular expression

**Code Generation:** Code generation can be considered as the final phase of compilation. A part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.

**Optimization:** Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

### [3.3] Design Rationale:

- a) We believe that the architecture of coolc should be as simple as possible, and should be as easy to understand as possible. The high level language compilers, argument serialization and deserialization scripts, storage data structure models.
- b) The syntax is very much stripped down, and the "standard library" contains only a few basic classes. Separate compilation is not supported, though the compiler does support multiple source files as input.
- c) Cool is built entirely on public domain tools; it generates code for a MIPS simulator, *spim*. Thus, the project should port easily to other platforms.

## 4. Data Design

### 4.1. DATA DESCRIPTION

#### Lexical Analysis:

File name	Description
cool.flex	This is the skeleton file for the specification of the lexical analyzer.
test.cl	This is the cool program that you can test the lexical analyzer on. It contains some errors, so it won't compile with coolc. However, test.cl does not exercise all lexical constructs of COOL.
cool-parse.h	Contains definitions that are used by almost all parts of the compiler.
stringtab_functions.{cc   h }	This file contains functions to manipulate the string tables.
utilities.{cc   h }	Contains functions used by the main () part of the lextest program.
lextest.cc	This file contains the main function which will call your lexer and print out the token that it returns.

mycoolc	Is a shell script that binds together the phases of compiler using Unix pipe. This architecture makes it easy to mix and match the components for the compiler designing.
cool-lexer.cc	It is the scanner generated by flex from cool.flex.
*.d	Autogenerates Makefiles that capture dependencies between source and header files in this directory.

## **PARSER:**

cool.y	Is the skeleton for the parser specification that you are to write.
Good.cl , bad.cl	Test a few features of the grammar
cool-tree.aps	Contains the definitions for the tree language which you use to construct the abstract syntax tree (AST).
tree.{cc h}	Contain definitions used by the tree package.
tokens-lex.cc	Is a lexer capable of reading a token stream from console in the format produced by the lexer phase
parser-phase.cc	Contains a driver to test the parser.
dumptype.cc	Prints the AST out in a form readable by the semant phase of the compiler.
handle_flags.cc	Implements routines for parsing

	command line flags
--	--------------------

## 4.2. DATA DICTIONARY

This section formally defines the type rules of Cool. The type rules define the type of every Cool expression in each context. The context is the type environment, which describes the type of every unbound identifier appearing in an expression.

```

program ::= [[class; ]]+
class    ::= class TYPE [inherits TYPE] { [feature; ] * }
feature  ::= ID( [ formal [[, formal]] ] ) : TYPE { expr }
          | ID : TYPE [ <- expr ]
formal   ::= ID : TYPE
expr     ::= ID <- expr
          | expr[@TYPE].ID( [ expr [[, expr]] ] )
          | ID( [ expr [[, expr]] ] )
          | if expr then expr else expr fi
          | while expr loop expr pool
          | { [[expr; ]]+}
          | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]] in expr
          | case expr of [[ID : TYPE => expr; ]]+esac
          | new TYPE
          | isvoid expr
          | expr + expr
          | expr - expr
          | expr expr
          | expr / expr
          | ~expr
          | expr < expr
          | expr <= expr
          | expr = expr
          | not expr
          | (expr)
          | ID

```



- | integer
- | string
- | true
- | false

- **FLEX:** Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). It's used as the lex implementation together with Yacc parser generator. Flex, an automatic lexical analyser, is often used with Bison, to tokenize input data and provide Bison with tokens.
- **BISON:** GNU bison, commonly known as Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser that reads sequences of tokens and decides whether the sequence follows the syntax specified by the grammar. Bison by default generates LALR parsers but can also create GLR parsers. It is compatible with yacc, but also has several extensions over this earlier program
- **References on MIPS & SPIM:** In our project, we use SPIM Simulator. spim is a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for this processor. spim also provides a simple debugger and minimal set of operating system services. spim does not execute binary (compiled) programs.

## 5. COMPONENT DESIGN

### Lexical Analysis Design:

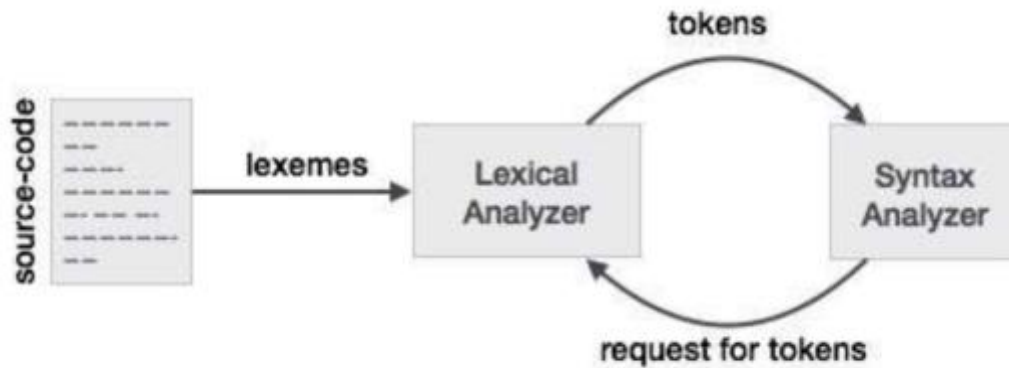


FIG 5. Lexical Analysis Design flow

### ALGORITHM:

```
DIGIT [0-9]
INTEGER {DIGIT}+
ESCAPE  \\
NEWLINE  \n
NULL_CHAR \0
ONE_CHAR_TOKENS  [ :+\\-*/=) ( ) { ~ . , ; < @ ]
TRUE      t(?i:rue)
FALSE f(?i:alse)
LPAREN    \(
RPAREN    \)
STAR      \*
ALPHANUM  [a-zA-Z0-9_]
TYPE_ID   [A-Z]{ALPHANUM}*
OBJECT_ID [a-z]{ALPHANUM}*
QUOTE     \"
HYPHEN    -
WHITESPACE [ \t\r\f\v]

DARROW    =>

%x COMMENTS COMMENT_IN_LINE STRING

%%

(?i:class) return CLASS;
(?i:else)  return ELSE;
(?i:fi)    return FI;
```

```
(?i:if)          return IF;
(?i:in)          return IN;
(?i:inherits)    return INHERITS;
(?i:let)         return LET;
(?i:loop)        return LOOP;
(?i:pool)        return POOL;
(?i:then)        return THEN;
(?i:while)       return WHILE;
(?i:case)        return CASE;
(?i:esac)        return ESAC;
(?i:of)          return OF;
(?i:new)         return NEW;
(?i:isvoid)      return ISVOID;
(?i:not)         return NOT;
"<="            return LE;
"<-"           return ASSIGN;
```

**Parsing Design and algorithm** is in progress.

## 6. HUMAN INTERFACE DESIGN

### 6.1. Overview of User Interface

We have used Linux environment. For lexical analysis we have used Flex and for Parser we plan to use Bison.

**To compile:**

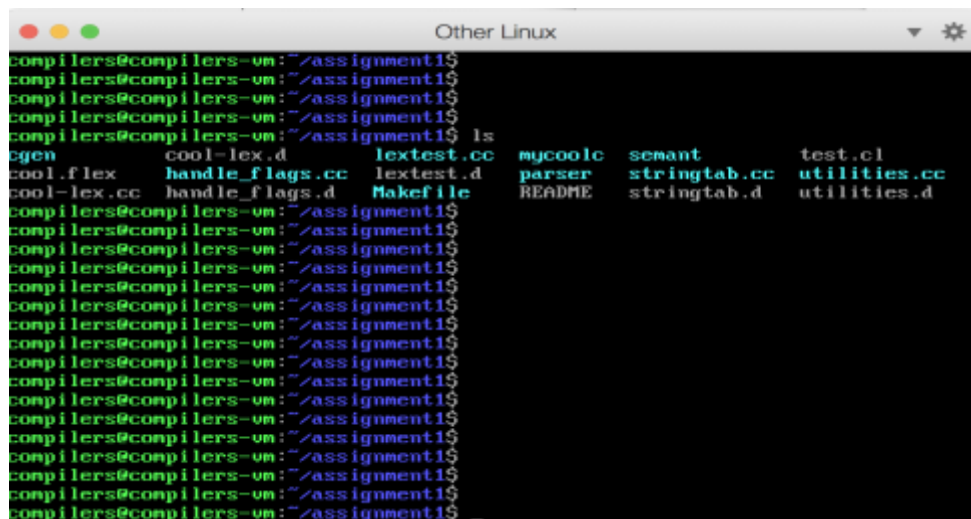
- % make lexer

**Put test input in 'helloworld.cl' and run the lexer code**

- % ./lexer helloworld.cl
- 

**To run your lexer on file test.cl**

- % ./mycoolc hello\_world.cl
- % helloworld.s



```
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$ ls  
cgen          cool-lex.d      lextest.cc     mycoolc       semant         test.cl  
cool.flex     handle_flags.cc lextest.d      parser        stringtab.cc  utilities.cc  
cool-lex.cc   handle_flags.d  Makefile       README       stringtab.d   utilities.d  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$  
compilers@compilers-vm:~/assignment1$
```

FIG 6. Linux environment showing the User Interface

## 6.2. Screen Images

### Code:

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Representation of occurrence of symbols using regular expressions defined in our code are as follows:

delim        [ \t\r\f\v]

ws            {delim}+

lower        [a-z]

upper        [A-Z]

letter        [a-zA-Z]

digit         [0-9]

Representation of language tokens using regular expressions defined in our code are as follows:

typeId        {upper}({letter}|{digit}|\_)\*

objectId      {lower}({letter}|{digit}|\_)\*

digits          {digit}+

**Instructions:**

## To compile:

```
% make lexer
```

Put test input in 'helloworld.cl' and run the lexer code

```
./lexer helloworld.cl
```

```
compilers@compilers-vm:~/Compiler/PA2$ make lexer
flex -d -ocool-lex.cc cool.flex
/bin/sh -ec 'g++ -MM -I. -I/usr/class/csl43/cool/include/PA2 -I/usr/class/csl43/cool/src/PA2 cool-lex.cc | sed '\''s/\(cool-lex\)\.o\] :|*/\1 stringtab.d :/g\'\'\' > cool-lex.d'
/bin/sh -ec 'g++ -MM -I. -I/usr/class/csl43/cool/include/PA2 -I/usr/class/csl43/cool/src/PA2 handle_flags.cc | sed '\''s/\(handle_flags\)\.o\] :|*/\1 utilities.d :/g\'\'\' > handle_flags.d'
/bin/sh -ec 'g++ -MM -I. -I/usr/class/csl43/cool/include/PA2 -I/usr/class/csl43/cool/src/PA2 stringtab.cc | sed '\''s/\(stringtab\)\.o\] :|*/\1 utilities.d :/g\'\'\' > stringtab.d'
/bin/sh -ec 'g++ -MM -I. -I/usr/class/csl43/cool/include/PA2 -I/usr/class/csl43/cool/src/PA2 utilities.cc | sed '\''s/\(utilities\)\.o\] :|*/\1 lextest.d :/g\'\'\' > lextest.d'
g++ -g -Wall -Wno-unused -Wno-write-strings -I. -I/usr/class/csl43/cool/include/PA2 -I/usr/class/csl43/cool/src/PA2 -c cool-lex.cc
g++ -g -Wall -Wno-unused -Wno-write-strings -I. -I/usr/class/csl43/cool/include/PA2 -I/usr/class/csl43/cool/src/PA2 lextest.o utilities.o stringtab.o handle_flags.o cool-lex.o -lfl -o lexer
compilers@compilers-vm:~/Compiler/PA2$
```

**FIG 7. Put test input in 'helloworld.cl' and run the lexer code**

```
compilers@compilers-vm:~/Compiler/PA2$ vim hello_world.cl
compilers@compilers-vm:~/Compiler/PA2$ ./lexer hello_world.cl
#name "hello_world.cl"
#1 CLASS
#1 TYPEID Main
#1 INHERITS
#1 TYPEID IO
#1 '{'
#2 OBJECTID main
#2 '('
#2 ')'
#2 ':'
#2 TYPEID SELF_TYPE
#2 '{'
#3 OBJECTID out_string
#3 '('
#3 STR_CONST "Hello, World.\n"
#3 ')'
#4 '}'
#4 ';'
#5 '}'
#5 ';'
compilers@compilers-vm:~/Compiler/PA2$
```

### FIG 8. Test the lexer code

To run your lexer on file test.cl

% ./mycoolc hello\_world.cl

% helloworld.s

A screenshot of a terminal window with a dark background and light-colored text. The text shows a sequence of commands and their outputs in a shell environment. The prompt is 'compilers@compilers-vm:~/Compiler/PA2\$'. The first command is './mycoolc hello\_world.cl', which produces several lines of output including version information and a copyright notice. The second command is 'spim helloworld.s', which outputs 'Hello, World.' and execution statistics. The final line shows the prompt again, indicating the session is still active.

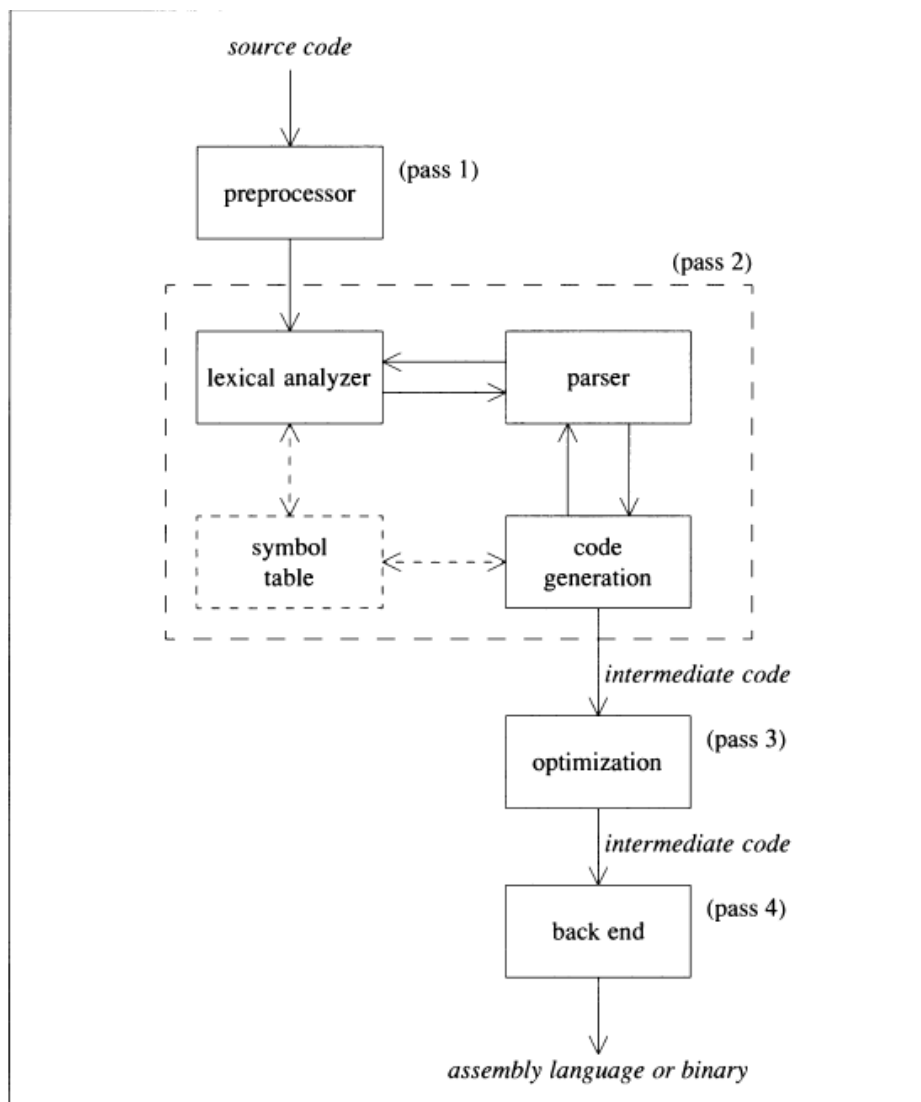
```
compilers@compilers-vm:~/Compiler/PA2$ ./mycoolc hello_world.cl
compilers@compilers-vm:~/Compiler/PA2$ spim helloworld.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/class/cs143/cool/lib/trap.handler
Hello, World.
COOL program successfully executed
Stats -- #instructions : 152
        #reads : 27  #writes 22  #branches 28  #other 75
compilers@compilers-vm:~/Compiler/PA2$
```

**FIG 9. Run the lexer code**

## 7. REQUIREMENTS

Compiler is a program that can read a program in one language i.e. the *source* language and translates it into equivalent program in another language i.e. the *target* language. If the target program is an executable machine language program, it can then be called by the user to process the inputs and produce the output.

Functional Requirements:



**FIG 10. Structure of typical four-pass compiler**



### Lexical Analyzer:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace in the source code.

### Parsing:

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing. We aim in following a top-down approach.

### Semantic Analysis:

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

### Code generation:

The Code Generated by the compiler is an object code of some lower-level programming language, for example, assembly language.

### Non-Functional:

#### Optimization:

Optimization can be assumed as something that removes unnecessary code lines, arranges sequence of statement to speed up the program execution without wasting resources. The compiler tries to minimize the time taken to execute a program by minimizing or maximizing some attributes of the executable computer program.

## 8. APPENDICES

### [A] Code for Lexical Analysis:

```
/*
 * The scanner definition for COOL.
 */

%{
#include <cool-parse.h>
#include <stringtab.h>
#include <utilities.h>

/* The compiler assumes these identifiers. */
#define yylval cool_yylval
#define yylex cool_yylex

/* Max size of string constants */
#define MAX_STR_CONST 1025
#define YY_NO_UNPUT /* keep g++ happy */

extern FILE *fin; /* we read from this file */

/* define YY_INPUT so we read from the FILE fin:
 * This change makes it possible to use this scanner in
 * the Cool compiler.
 */
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
    if ( (result = fread( (char*)buf, sizeof(char), max_size, fin)) < 0) \
        YY_FATAL_ERROR( "read() in flex scanner failed");

extern int curr_lineno;
extern int verbose_flag;

extern YYSTYPE cool_yylval;

static int comments_stack;
static int null_char_present;
static std::string current_string;

%}

DIGIT [0-9]
INTEGER {DIGIT}+
ESCAPE \\
```

```

NEWLINE      \n
NULL_CHAR    \0
ONE_CHAR_TOKENS  [:+\-*/=) (){~.,;<@]
TRUE         t(?i:rue)
FALSE        f(?i:alse)
LPAREN       \(
RPAREN       \)
STAR         \*
ALPHANUM     [a-zA-Z0-9_]
TYPE_ID      [A-Z]{ALPHANUM}*
OBJECT_ID    [a-z]{ALPHANUM}*
QUOTE        \"
HYPHEN       -
WHITESPACE   [ \t\r\f\v]

```

```
DARROW      =>
```

```
%x COMMENTS COMMENT_IN_LINE STRING
```

```
%%
```

```

(?i:class)  return CLASS;
(?i:else)   return ELSE;
(?i:fi)     return FI;
(?i:if)     return IF;
(?i:in)     return IN;
(?i:inherits) return INHERITS;
(?i:let)    return LET;
(?i:loop)   return LOOP;
(?i:pool)   return POOL;
(?i:then)   return THEN;
(?i:while)  return WHILE;
(?i:case)   return CASE;
(?i:esac)   return ESAC;
(?i:of)     return OF;
(?i:new)    return NEW;
(?i:isvoid) return ISVOID;
(?i:not)    return NOT;
"<="       return LE;
"<-"       return ASSIGN;

```

```
{WHITESPACE} ;
```

```

{HYPHEN}{HYPHEN} {
    BEGIN COMMENT_IN_LINE;

```

```
}

{DARROW}          {return (DARROW); }

{ONE_CHAR_TOKENS} {
    return int(yytext[0]);
}

{NEWLINE}    curr_lineno++;

{TRUE}       {
    cool_yylval.boolean = 1;
    return BOOL_CONST;
}

{FALSE} {
    cool_yylval.boolean = 0;
    return BOOL_CONST;
}

<COMMENTS>. ;

%%
```





























