Compiler Project Report

Prepared for

Dr. Eghbal Mansouri

Prepared by

Ali Maher

alimohammadmaher@gmail.com

9th of Jan 2023

Introduction

The project is supposed to read a calculation expression and produce a three-address code in C Language.

The Expressions contain integers and decimals, variables, parentheses, white spaces, and the following operations:

- Summation ➔ :A:
- Subtraction ➔ :S:
- Multiplication ➔ :M:
- Division ➔ :D:

Examples

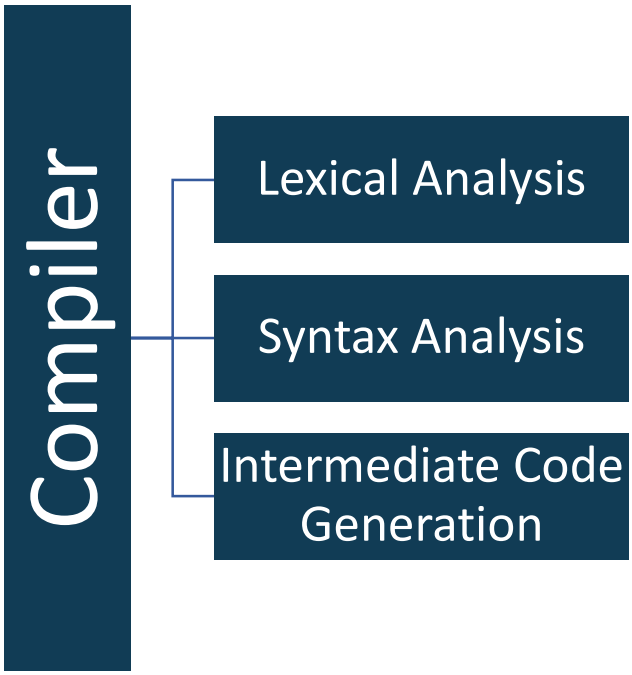
x = 3 :D: 4:S: 12	t1 = 4/3; x = 12-t1;
x = 3 :D: (4:S: 12)	t1 = 12-4; x = t1/3;

x = 36 :M: test :D: ps :A:123	t1 = ps/test;
	t2 = t1*36;
	x = 123+t2;

The priority of the operators is as usual, but their participation is the opposite.

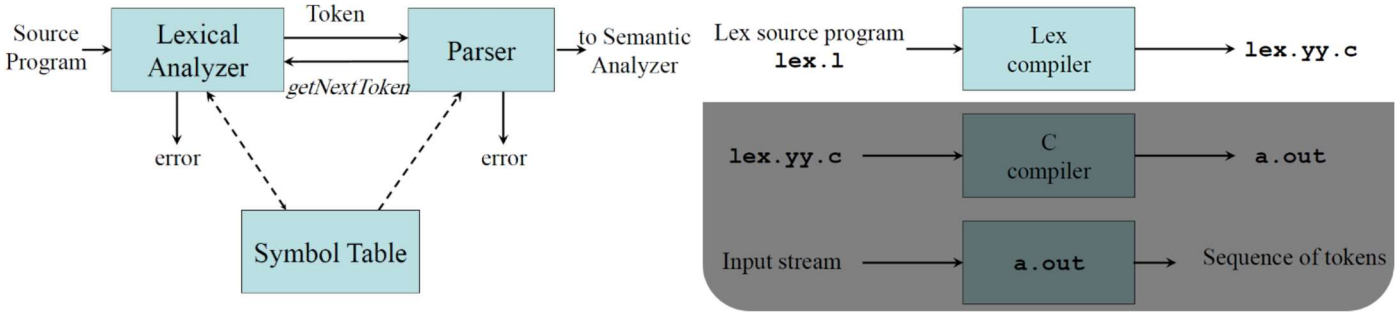
For each operator, the calculation of the operands is reversed, that means, the right and left operands are shifted and calculated.

The project has been divided into three phases.



Phase 1: Lexical Analysis

In this phase, we have a string of characters that we had to convert to tokens. As you can see, we give a source program to the Lexical Analyzer and receive Tokens as output.



The input is a Lex file with (.l) suffix, and the output is a file with (.c) suffix.

- **What is LEX?**

  It is a tool or software which automatically generates a lexical analyzer (finite Automata). It takes as its input a LEX source program and produces lexical Analyzer as its output. Lexical Analyzer will convert the input string entered by the user into tokens as its output. (More info)

First of all, we need to download and install Lex and Yacc. So, I've got some help from this site. Then we can use them easily like below:

- **Compiling and Running LEX Programs:**
  (Note: hello.l is the LEX program)
  flex hello.l
  gcc lex.yy.c (y.tab.c)
  a.exe

- **Compiling and Running YACC Programs:**
  (Note: hello.l is the LEX and hello.y is the program)
  flex hello.l
  bison -dy hello.y
  gcc lex.yy.c y.tab.c
  a.exe

Now let's write our Lex file. A Lex specification consists of three parts:

Regular definitions, C declarations in %{ %}

%%

Translation rules

%%

User-defined auxiliary procedures

```
%{
#include <stdio.h>
%}
digit      [0-9]
letter     [A-Za-z]
id         {letter}({letter}|{digit})*
%%
{digit}+   { printf("number: %s\n", yytext); }
{id}       { printf("ident: %s\n", yytext); }
.          { printf("other: %s\n", yytext); }
%%
main() {
  yylex();
}
```

Regular definition

Translation rules

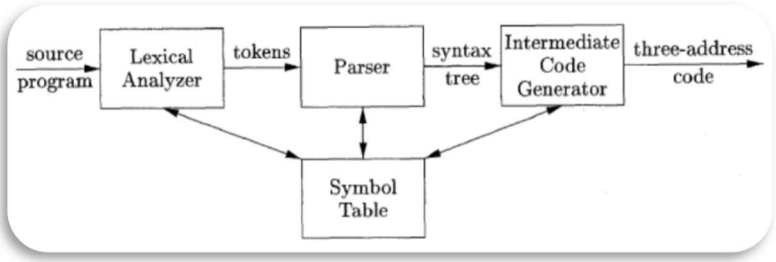
In the Regular definition part, we define some definitions with names "add", "sub", "mul", and "div" with the following values: ":A:", ":S:", ":M:" and ":D:". These are our operations.

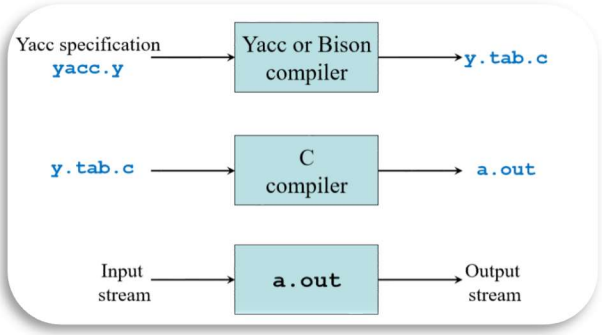
Also, we need to define "letter", "digit", "id", and "number".

Note that, in this phase, in action parts, we change some considered flags and then print them to make sure that it works correctly. For example, I consider "nNumber" which counts the number of numbers in our input string.

Phase 2: Parsing

In this phase, we should declare translation rules to make our language. We use the Yacc tool which is short form of "Yet another compiler-compiler". It's equivalent in Windows is Bison. The Yacc works based on LALR parser.





Now let's write our yacc file. Like Lex, Yacc specification consists of three parts:

Yacc declarations, C declarations in %{ and %}

%%

Translation rules

%%

User-defined auxiliary procedures

- **The translations rules are productions with actions:**

Production_1 { semantic action_1 }
Production_2 { semantic action_2 }
...
Production_n { semantic action_n }

- **Productions in Yacc are of the form:**

Nonterminal : tokens/nonterminals { action }
| tokens/nonterminals { action }
...
;

- **Tokens:**

Named tokens which used in Lex file must be declared first in the declaration part as:

%token TokenName

- **We also know that LALR parser is S-attributed, so:**

X : Y1 Y2 Y3 … Yn{ action }

– $$ refers to the value of the attribute of X – $i refers to the value of the attribute of Yi

For example:

factor : '(' expr ')' { $$=$2; }

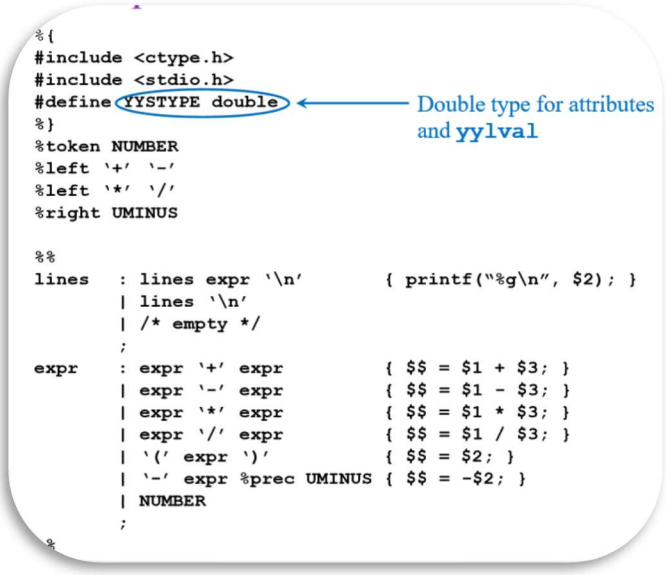
- **Precedence Levels and Associativity:**

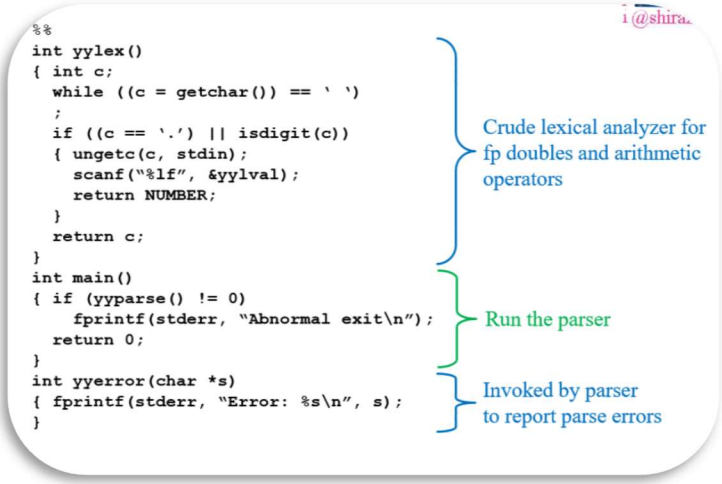
The "left" is used to say that the expression associativity is from left to right. And if we use "right" it means the associativity is from right to left.

The precedence levels defined by their lines' order.

%left '+' '-'
%left '' '/'*

- **Example:**





Phase 3: Generate Three-Address-Code

In this phase, three-address-code generated based on the parse trees. The point is that phase two and phase three cannot be separated. Actually, the parsing is with translation.

So, when we call "yyparse" in the Yacc file, the parsing and translating occur at the same time.

- **Combining Lex file with Yacc file:**

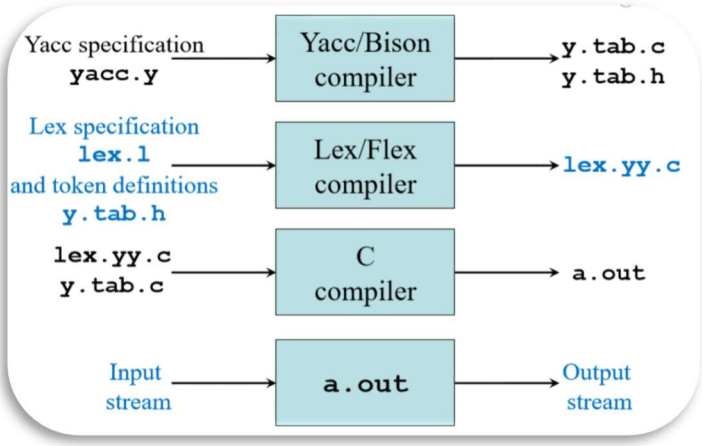
When we run the "yacc.y" with the Bison a header file get created that we should include it in our lex file. And for other common variables we should use "extern" keyword to declare it.

```
%option noyywrap
%{
#include "y.tab.h"

extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]             { /* skip blanks */ }
{number}        { sscanf(yytext, "%lf", &yylval);
                  return NUMBER;
                }
\n|.            { return yytext[0]; }
```

Generated by Yacc, contains `#define NUMBER xxx`

Defined in `y.tab.c`

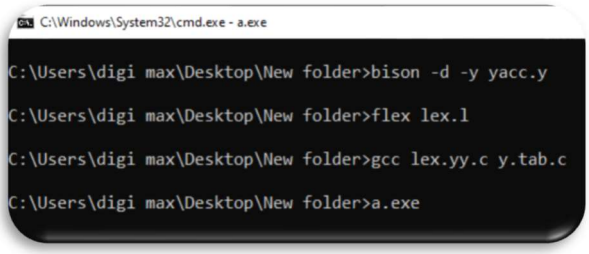


How to run the final project?

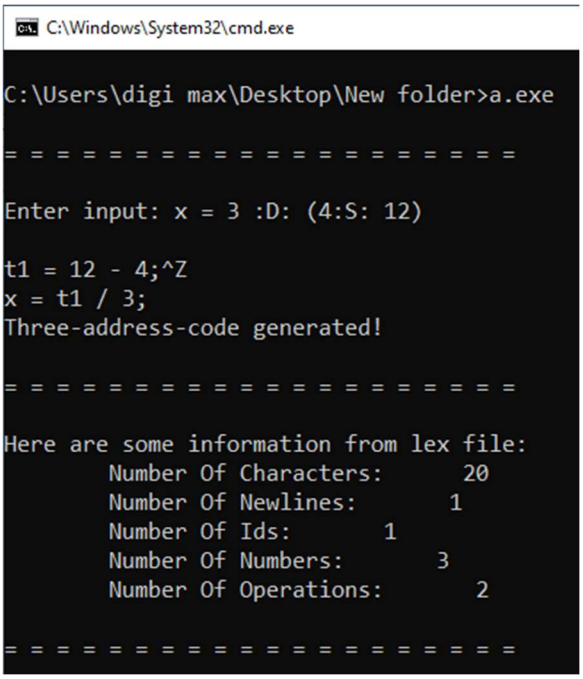
In windows we should run the following instructions to run the project.

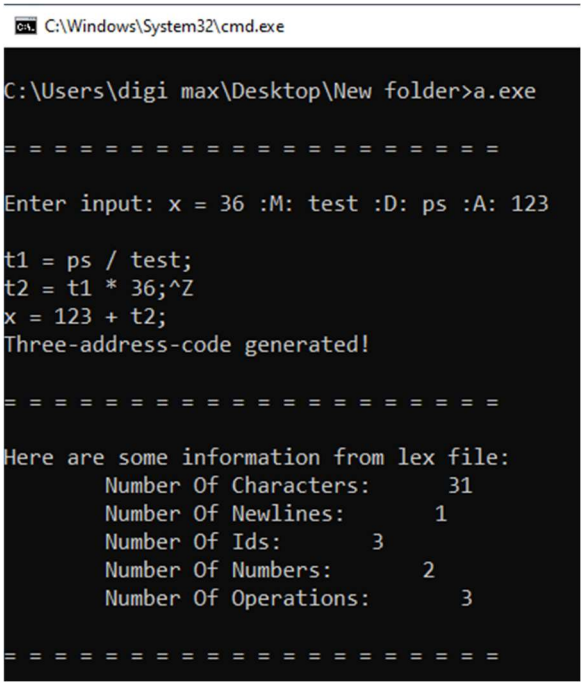
bison -d -y yacc.y
flex lex.l
gcc lex.yy.c y.tab.c
a.exe

Now the program is ready to use :)



- **Some outputs:**





Summary:

We create a simple compiler with simple instructions in three phases with Yacc and Lex tools.

More details are commented on the files. So, it would be easier to track.

The project files with doc are also available on my GitHub.

Thanks for your time