2CS701

COMPILER CONSTRUCTION PROJECT REPORT
MINI COMPILER IN C

Submitted in partial fulfillment of the requirements for semester-VII

Prepared by:

SHAILI PATEL - (18BCE168)
MANASWI PIPALIYA - (18BCE177)

Date:
18th November 2021

Department of computer science and engineering,
Institute of Technology,
Nirma University,
Ahmedabad, Gujarat.

INTRODUCTION

This project intends to create intermediate code for the language for specific constructs. Constructs like loops. For loop, while loop and also conditional statements. Collecting continuous characters to build valid tokens such as keywords, identifiers, constants, Special symbols, Operators like arithmetic operator,logical operator,etc to create a mini compiler for a language which will also include control statements.

Features Of This Mini Compiler

- This following mini c compiler works for the following constructs :
  - If
  - Else
  - Only if
  - For
  - Nested statements
- It also shows error like :
  - Showing error message for the variables that are redeclared
  - Checks return type mismatch

Process

A compiler is actually a collection of 3 - 4 components (with some subjugates) that are fed from one to another in a pipeline method. To assist us build out each of these components, we'll be employing a different tool. Here's a representation of each stage as well as the tool we'll be using:

Source: www.gnuu.org

Tools Used

- Flex - open source tool
- Bison - For semantic parsing, we'll be using Yacc, better known as Bison.
- Code editor

PROCEDURE

- Symbol Table
- Parse Tree
- Semantic Analysis
- Intermediate Code Generation

Phase-1

Lexical analysis

We must convert our input into a list of known tokens. As explained previously, our grammar has only the most fundamental tokens: identifiers, numerals (integer arithmetic and decimals), mathematical operators, parenthesis, and brackets. "*yylavl*" is used to preserve the output of each scanned lexeme. The lex var yytext holds the matching term.

```
struct dataType {
    char * id_name;
    char * data_type;
    char * type;
    int line_no;
} symbol_table[40];
```

Phase-2

Syntax Analysis

In syntax analysis, we declared a struct to represent the node in the binary tree that would be created. The struct node has two attributes: left and right, as well as a token that is a character array.To produce the syntax tree, a node is created for each token and linked to the nodes of the tokens that appear semantically to its left and right. The resulting syntax tree should be traversed in order to reconstruct the programme logically. Yacc tool is used which reports shift-reduce and reduce-reduce conflicts in given grammar.

Phase-3

Semantic Analysis

Three sorts of static tests are handled by the semantic analyzer.

- Variables must be stated before they may be used. The check declaration function is used to check if the identifier supplied as an argument is available in the symbol table. If it isn't, a helpful error message is displayed. When an identifier is met in a statement that isn't a declarative statement, the check declaration function is invoked.
- It is not possible to redeclare variables. Variables cannot be redeclared even throughout loops since our compiler expects a single scope. The add method has been modified to verify if the symbol is already existing in the symbol table before inserting it. Error message is printed in case of redeclaration.

```
printf(errors[errors], "Line %d: many declarations of \"%s\"  are not allowed!\n", countn+1, yytext);
errors++;
```

- Variables in an arithmetic expression are type checked. Nothing is done if the types match. When a variable has to be changed to a different type, a type conversion node is added to the syntax tree. To keep track of the type of complex expressions that aren't in the symbol table, the type field is added to the struct representing value and expression tokens. The annotated syntax tree is the result of this phase.

```
void checkreturn_type(char *value) {
    char *main_datatype = get_type("main");
    char *return_datatype = get_type(value);
    if((!strcmp(main_datatype, "int") && !strcmp(return_datatype, "CONST")) || !strcmp(main_datatype, return_datatype)){
        return ;
    }
    else {
        sprintf(errors[errors], "Line %d: Return type mismatch\n", countn+1);
        errors++;
    }
}
```

Phase 4

Intermediate Code Generation

The three-address-code representation is applied in this case. Variables were utilised to monitor the generation of the following temporary variable and label. The labels to travel to in case the condition is satisfied or not satisfied were also indicated in the if and for condition statements. The intermediate code is the result of this stage.

The following lex file:

```
%%

"int"              { return INT; }
"float"            { return FLOAT; }
"char"             { return CHAR; }
"void"             { return VOID; }
"return"           { return RETURN; }
"for"              { return FOR; }
"if"               { return IF; }
"else"             { return ELSE; }
"scanf"            { return SCANFF; }
"printf"           { return PRINTFF; }
^"#include"[ ]*<.+\.h>     { return INCLUDE; }
```

```
"true"                  { return TRUE; }
"false"                 { return FALSE; }
"<="                    { return LE; }
">="                    { return GE; }
"=="                    { return EQ; }
"!="                    { return NE; }
">"                     { return GT; }
"<"                     { return LT; }
"&&"                    { return AND; }
"||"                    { return OR; }
"+"                     { return ADD; }
"-"                     { return SUBTRACT; }
"/"                     { return DIVIDE; }
"*"                     { return MULTIPLY; }
[-]?{digit}+            { return NUMBER; }
[-]?{digit}+\.{digit}{1,6}  { return FLOAT_NUM; }
{alpha}({alpha}|{digit})*   { return ID; }
{unary}                 { return UNARY; }
\/\/.*                  { ; }
\/\*(.*\n)*.*\*\/       { ; }
[ \t]*                  { ; }
[\n]                    { countn++; }
.                       { return *yytext; }
["].*["]                { return STR; }
['].[']                 { return CHARACTER; }

%%
```

The following is our CFG:

```
program: headers main '(' ')' '{' body return '}'
;

headers: headers headers
| INCLUDE
;

main: datatype ID
;

datatype: INT
| FLOAT
| CHAR
| VOID
;

body: FOR '(' statement ';' condition ';' statement ')' '{' body '}'
| IF '(' condition ')' '{' body '}' else
| statement ';'
```

```
| body body
| PRINTFF '(' STR ')' ';'
| SCANFF '(' STR ',' '&' ID ')' ';'
;

else: ELSE '{' body '}'
|
;

condition: value relop value
| TRUE
| FALSE
;

statement: datatype ID init
| ID '=' expression
| ID relop expression
| ID UNARY
| UNARY ID
;

init: '=' value
|
;

expression: expression arithmetic expression
| value
;

arithmetic: ADD
| SUBTRACT
| MULTIPLY
| DIVIDE
;

relop: LT
| GT
| LE
| GE
| EQ
| NE
;

value: NUMBER
| FLOAT_NUM
| CHARACTER
| ID
;
```

```
return: RETURN value ';'
|
;

%%
```

TO RUN THE .L & .Y FILE :

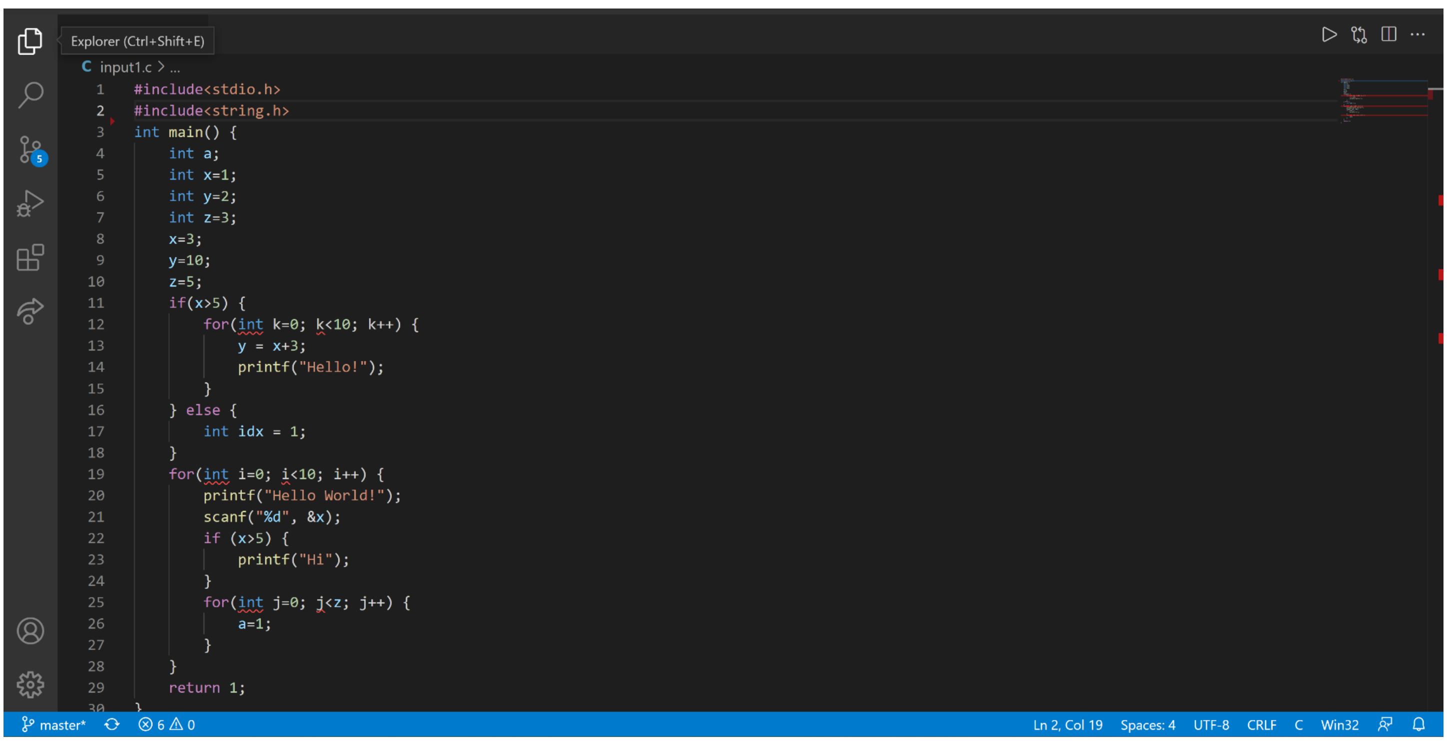
To install flex and bison & setup in windows machine: [link]

To run the code write the following commands in the command prompt:

```
flex lexer.l
bison -yd parser.y
gcc -lm y.tab.c -std=c99 -w
a< input1.c
```

TEST CASES

Input 1



Stage-1

```
Mini compiler by 18bce168 & 18bce177
******stage 1******

symbol   datatype   type    location
_____________________________________

#include<stdio.h>                Header  0
#include<string.h>               Header  1
main     int     Function        3
a        int     Variable        4
x        int     Variable        5
1        CONST   Constant        5
y        int     Variable        6
2        CONST   Constant        6
z        int     Variable        7
3        CONST   Constant        7
10       CONST   Constant        9
5        CONST   Constant        10
if       N/A     Keyword         11
for      N/A     Keyword         12
k        int     Variable        12
0        CONST   Constant        12
printf   N/A     Keyword         14
else     N/A     Keyword         16
idx      int     Variable        17
i        int     Variable        19
scanf    N/A     Keyword         21
j        int     Variable        25
return   N/A     Keyword         29
```

Stage-2

For the syntax analysis phase, we can show the inorder traversal of the parse tree.

```
******stage 2******:


the parse Tree is:

#include<stdio.h>,
 headers,
 #include<string.h>,
 program,
 a,
 declaration,
 NULL,
 statements,
 x,
 declaration,
 1,
 statements,
 y,
 declaration,
 2,
 statements,
 z,
 declaration,
 3,
 statements,
 x,
 =,
 3,
 statements,
 y,
 =,
 10,
 statements,
 z,
 =,
 5,
 statements,
 x,
 >,
```

Stage 3: since there were no semantic errors, it shows no errors in the code message.

```
******stage 3******

no errors in the code
```

Stage 4:

```
******stage 4******

a = NULL
x = 1
y = 2
z = 3
x = 3
y = 10
z = 5

if (x > 5) GOTO L0 else GOTO L1

LABEL L0:
k = 0

LABEL L2:

if NOT (k < 10) GOTO L3
t1 = x + 3
y = t1
t1 = k + 1
k = t0
JUMP to L2

LABEL L3:

LABEL L1:
idx = 1
GOTO next
i = 0

LABEL L4:

if NOT (i < 10) GOTO L5

if (x > 5) GOTO L6 else GOTO L7
```

Input 2

```c
#include<stdio.h>
#include<string.h>

int main() {
    int i=1;
    float f = 2.5;
    char c = 'A';
    int x = 3.5;
    i = x + f * c;
    return 3;
}
```

```
Mini compiler by 18bce168 & 18bce177
******stage 1******

symbol   datatype   type   location
_______________________________________

#include<stdio.h>               Header  0
#include<string.h>              Header  1
main    int     Function        3
i       int     Variable        4
1       CONST   Constant        4
f       float   Variable        5
2.5     CONST   Constant        5
c       char    Variable        6
'A'     CONST   Constant        6
x       int     Variable        7
3.5     CONST   Constant        7
return  N/A     Keyword         9
3       CONST   Constant        9
```

```
******stage 3******

no errors in the code
******stage 4******

i = 1
f = 2.5
c = 'A'
x = 3.5
t0 = f * c
t1 = x + t0
i = t1
```

Input 3

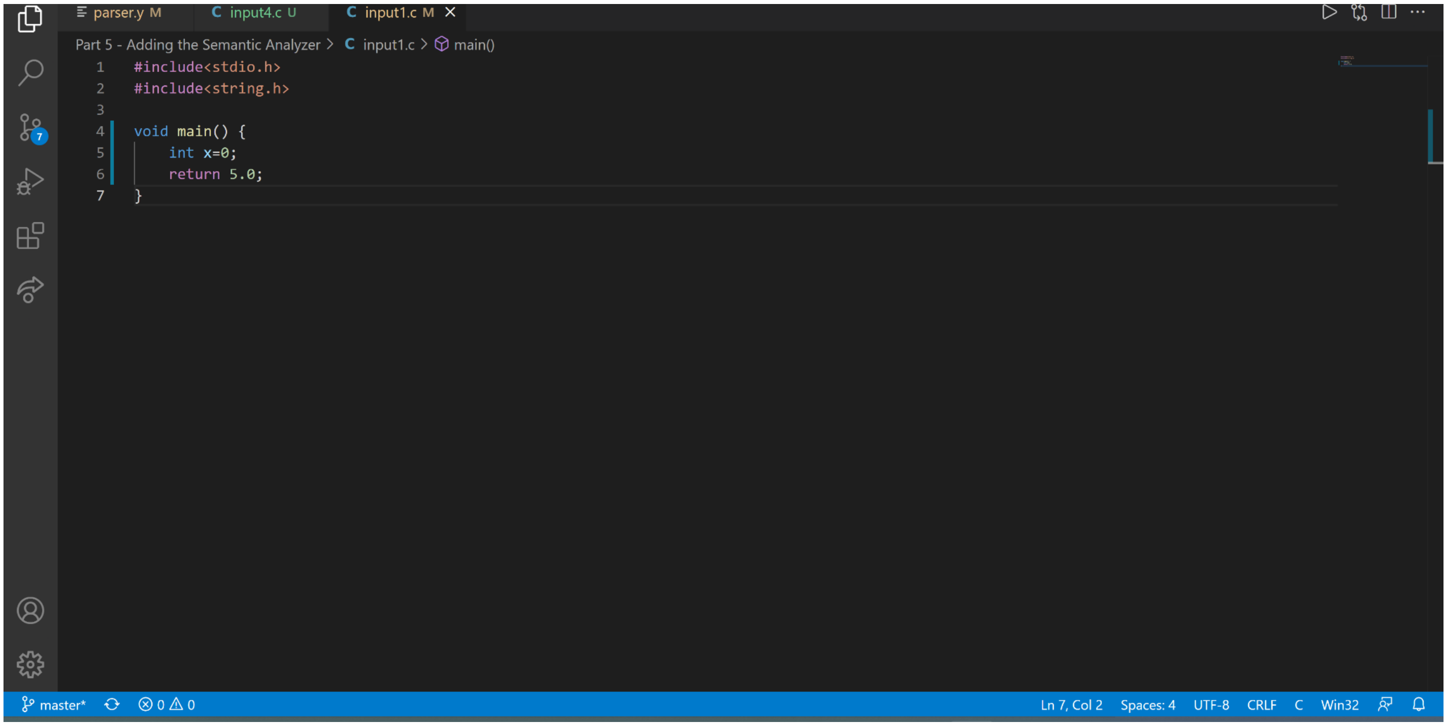
In this input we will see that the phase-3 will give us errors regarding redeclaration of variables.

```c
1   #include<stdio.h>
2   #include<string.h>
3
4   int main() {
5       int x=1;
6       float f;
7       int a=3;
8       int x;
9       a = x * 3 + 5;
10      if(x>a) {
11          printf("Hi!");
12          a = x * 3 + 100;
13          if(x>a) {
14              printf("Hi!");
15              a = x * 3 + 100;
16          }
17          else {
18              x = a * 3 + 100;
19          }
20      }
21      else {
22          x = a * 3 + 100;
23      }
24  }
```

```
******stage 3******

 1 errors
        - Line 8: many declarations of "x"  are not allowed!
```

Input 4



CONCLUSION

Thus different stages like lexical analysis, syntax and semantic analysis, Intermediate Code Generation helps in the formation of a mini compiler. The given compiler accepts assignments and arithmetic operations, loop statements and nested statements. The project is all about building a mini compiler from scratch. While building the compiler we learned various strategies to design and implement a successful compiler. Many more constructs like switch, while, do while could be added to the compiler. Also we can go a step further and automatically try to change the syntax like if a developer writes fro instead of for , the compiler can automatically change fro → for, this way the developer can save a lot of time debugging the code. Furthermore, we can implement code optimisation algorithms for faster and efficient compilations.

REFERENCES

- https://www.cs.cmu.edu/~aplatzer/course/Compilers/waitegoos.pdf
- https://www.admb-project.org/tools/flex/compiler.pdf
- https://gnuu.org/2009/09/18/writing-your-own-toy-compiler