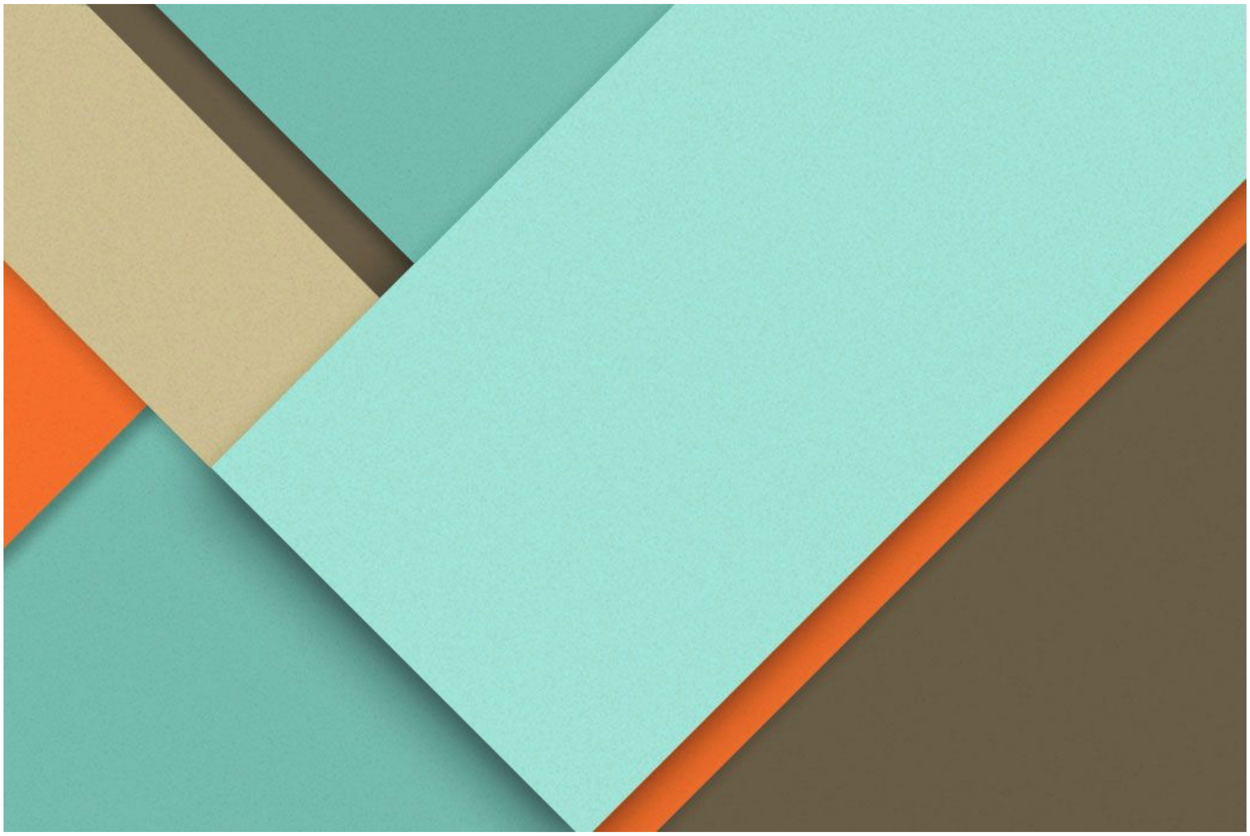


Compiler for C++ Grammar

4-4-2019

Sarthak Agrawal	160905420	65
Chhayank Jain	160905328	49
Vidisha Shah	160905278	39

Objective

For a given grammar pertaining to the C++ language,we use Flex and Bison to generate tokens which are then passed through the above grammar to accept or reject the string.

A hash table data structure is used for the implementation and error detection has been implemented too.

Features

I.Parsing constructs of C++

II.Token List Generation

III.Symbol Table Generation

IV.Error Reporting

Grammar

p_prime: program

program: elements mainFunction

elements: structs elements| functions_or_declarations elements| classes elements | eps

structs: STRUCT ID '{' declarations_only '}' ';'

classes: CLASS ID '{' specifier ':' class_body '}' ';'

specifier: private | public | protected

class_body: functions_or_declarations class_body | eps

functions_or_declarations: DTYPE ID next ';'

next: functions | declarations | eps

functions: '(' parameter_list ')' '{' declarations_only statement_list '}';

parameter_list: DTYPE ID id_prime | eps

id_prime: COMMA parameter_list| '[' NUMBER ']' id_prime | eps

mainFunction: MAIN '(' ')' '{' declarations_only statement_list '}';

declarations: identifier_list ';'| eps

declarations_only: DTYPE identifier_list ';' declarations_only | eps

identifier_list: ID id1;

id1: COMMA identifier_list |'[' NUMBER ']' id1| ;

statement_list: statement statement_list| ;

statement: assign_stat_or_function_call ';'| decision_stat | looping_stat| return_stat | input_stat | output_stat| switch_stat

assign_stat_or_function_call: ID assign_function_prime;

assign_function_prime: function_call|assign_stat;

function_call: '(' argument_list ')' | '.' ID f_prime

f_prime: '.' f_prime| function_call

argument_list: identifier_list| eps

assign_stat: '=' expression | '.' ID a_prime

a_prime: '.' a_prime | assign_stat

return_stat : RETURN factor ';'

input_stat:CIN '>''>' ID input_prime ';'

input_prime : '>' '>' ID input_prime| eps output_stat: COUT '<' '<' LITERAL output_prime ';' | COUT '<' '<' NUMBER output_prime ';' | COUT '<' '<' ID output_prime ';'

output_prime: '<' '<' LITERAL output_prime|'<' '<' ID output_prime|'<' '<' NUMBER output_prime| eps

assign_stat_only: ID assign_stat_prime

assign_stat_prime: '=' expression | '.' assign_stat_only

expression: simple_expression eprime;

eprime: relop simple_expression| eps

simple_expression: term seprime

seprime: addop term seprime| eps

term: factor tprime

tprime: mulop factor tprime| eps

factor: ID|NUMBER

decision_stat: IF '(' expression ')' '{' statement_list '}' dprime

dprime: ELIF '(' expression ')' '{' statement_list '}' dprime | dp_prime

dp_prime: ELSE '{' statement_list '}' | eps

switch_stat: SWITCH '(' ID ')' '{' case_stat '}'

case_stat: CASE NUMBER ':' statement_list break_stat case_stat | default | eps

break_stat: BREAK ';' | eps

default: DEFAULT ':' statement_list

looping_stat: WHILE '(' expression ')' '{' statement_list '}'| FOR '(' assign_stat_only ';' expression ';' assign_stat_only ')' '{' statement_list '}'

relop: '>'|'<'|'<' '='|'>' '=' | '!' '='

addop: '+'|'-'

mulop: '*' | '/' | '%'

Languages

I. C

The symbol table has been implemented using C language along with parts of the parser.

II. Flex

Tokens from the input string are generated using Flex

III. Bison

Bison is used the check the acceptance of the input by passing the tokens through the grammar.

Parser Type

The parser is a Top Down Parser

Methodology

The input file is read and the text is passed through the grammar in the Bison file.For any terminals,generated tokens are then passed back using the Flex file.In the rules of the productions,special care is taken to record the data type of the identifiers that might be placed in the symbol table.These identifiers are then searched through the symbol table using the hash key generated through the hash function and is placed in the appropriate node.

If the input conforms to the grammar,then the string is accepted and the symbol table is printed.

Otherwise the parsing stops and the location of the Error is demonstrated.

User Manual

1.Place all files under a single directory and set path to the directory.

2.Compile bison by calling ***$ bison -d parser.y***

3.Compile Flex by calling ***$ flex parser.l***

4.Call ***$ gcc lex.yy.c parser.tab.c -o parser.o***

5.Call the output file ***$ ./parser.o***

CODE

parser.l

```
%{
   #include "parser.tab.h"
   #include "symTable.h”
%}

%option yylineno

%%
("/*"(.|\n)*"*/") {;}
("//"(.)*) {;}

[0-9]+ {printf("%s\n",yytext);return NUMBER;}

"int"|"char"|"void"|"string" {printf("%s\n",yytext);strcpy(dtype,yytext);return DTYPE;}
```

```
"cout" {printf("%s\n",yytext);return COUT;}
"cin" {printf("%s\n",yytext);return CIN;}
\"(.)*\" {printf("%s\n",yytext);return LITERAL;}
"main" {printf("%s\n",yytext);return MAIN;}
"class" {printf("%s\n",yytext);strcpy(dtype,yytext);return CLASS;}
"struct" {printf("%s\n",yytext);strcpy(dtype,yytext);return STRUCT;}
"private"|"public"|"protected" {printf("%s\n",yytext); return SPECIFIER;}
"while" {printf("%s\n",yytext); return WHILE;}
"for" {printf("%s\n",yytext); return FOR;}
"if" {printf("%s\n",yytext); return IF;}
"else if" {printf("%s\n",yytext); return ELIF;}
"else" {printf("%s\n",yytext); return ELSE;}
"switch" {printf("%s\n",yytext); return SWITCH;}
"case" {printf("%s\n",yytext); return CASE;}
```

```
"break" {printf("%s\n",yytext);return BREAK;}

"default" {printf("%s\n",yytext); return DEFAULT;}

"return" {printf("%s\n",yytext);return RETURN;}

[a-zA-Z][a-zA-Z0-9_]* {printf("%s \n",yytext);
        int index = searchTable(yytext);
        if(index==-1) {
                TOKEN* tk = newToken();
                index = hashFunction();
                tk->index = index;
                strcpy(tk->name,yytext);
                strcpy(tk->datatype,dtype);
                if(strcmp(dtype,"class")==0 || strcmp(dtype,"struct")==0) {
                        tk->scope=0;
                } else {
                        tk->scope=1;
                }
                SYMTABLE[index]=tk;
        }
        return ID;
}

"," {printf("%s\n",yytext); return COMMA;}
" "|\t|\n {;}
. {return yytext[0];}
%%
```

```
int yywrap() {
    return 1;
}
```

parser.y

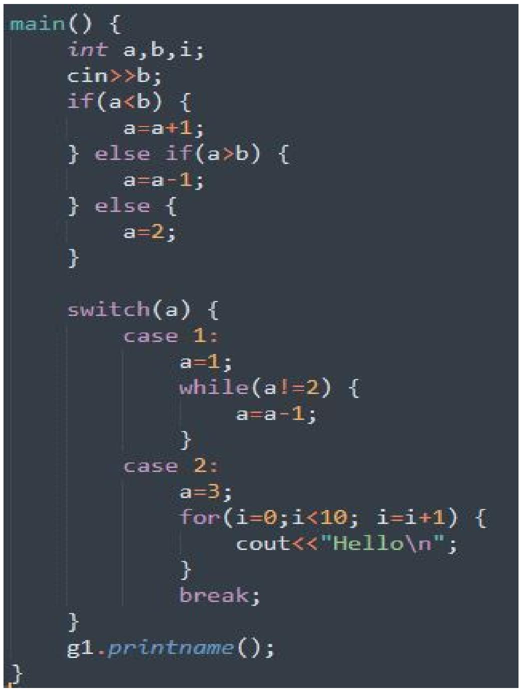
```
%{
    #include <stdio.h>
    #include <stdlib.h>
    extern FILE *yyin;
    int yyerror();
    int yylex();

    void printSymbolTable();
    int yylineno;
%}
%token STRUCT ID CLASS SPECIFIER MAIN DTYPE RETURN CIN COUT NUMBER IF ELIF ELSE WHILE FOR LITERAL BREAK DEFAULT COMMA SWITCH CASE
CLASSNAME STRUCTNAME

%%
relop: '>'|'<'|'<' '='|'>' '=' | '!' '=';
addop: '+'|'-';
mulop: '*' | '/' | '%';

%%

int yyerror(char *msg) {
        printf("\n\nSyntax Error -");
    printf("Invalid Expression at line %d\n",yylineno);
    exit(0);
}

void main() {
    yyin = fopen("input.c","r");
    yyparse();
}
```

symtab.h

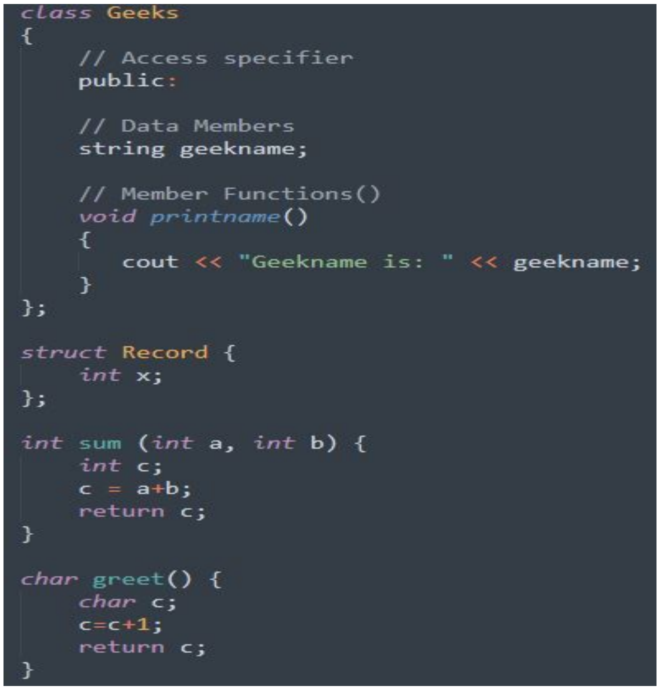
```
#ifndef symTable_h
#define symTable_h
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TABLE_SIZE 20

int hashIndex=0;

typedef struct {
        int index;
        char name[10];
        int scope;
        char datatype[10];
        int isFunction;
        char returnType[10];
        int argCount;
        int* argsID;
}TOKEN;

TOKEN* SYMTABLE[TABLE_SIZE];
```

```
TOKEN* newToken() {
	TOKEN* temp = (TOKEN*)malloc(sizeof(TOKEN));
	return temp;
}
TOKEN* initialiseArgs(TOKEN* f, int argCount) {
	f->argCount = argCount;
	f->argsID = (int*)calloc(argCount,sizeof(int));
	int i;
	for(i=0; i<argCount; i++) {
		f->argsID[i]=-1;
	}
}


int hashFunction() {
	return hashIndex++;
}

void initialiseTable() {
	int i;
	for(i=0; i<TABLE_SIZE; i++) {
		SYMTABLE[i] = NULL;
	}
}

void printToken(TOKEN* t) {
	printf("%-12d%-12s%-12s",t->index,t->name,t->datatype);
	t->scope==1?printf("\tL\n"): printf("\tG\n");
}

void printSymbolTable() {
    int i;
    printf("--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--x--\n");
    printf("\t\t\tSYMBOL TABLE\n\n");
    printf("index\tlexemeName\tDatatype\tscope\n\n");
    for(i=0; i<TABLE_SIZE; i++) {
        if(SYMTABLE[i]==NULL)
            continue;
        printToken(SYMTABLE[i]);
    }
}

int searchTable(char* item) {
    int i;
    for(i=0; i<TABLE_SIZE; i++) {
        if(SYMTABLE[i]!=NULL && strcmp(SYMTABLE[i]->name,item)==0) return i;
    }
    return -1;
}
#endif
```

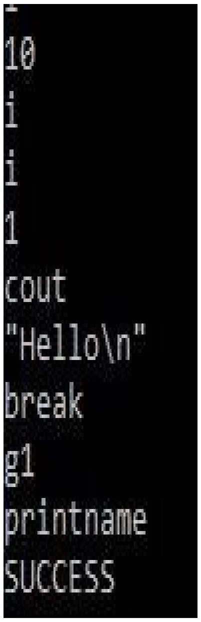
Snaps



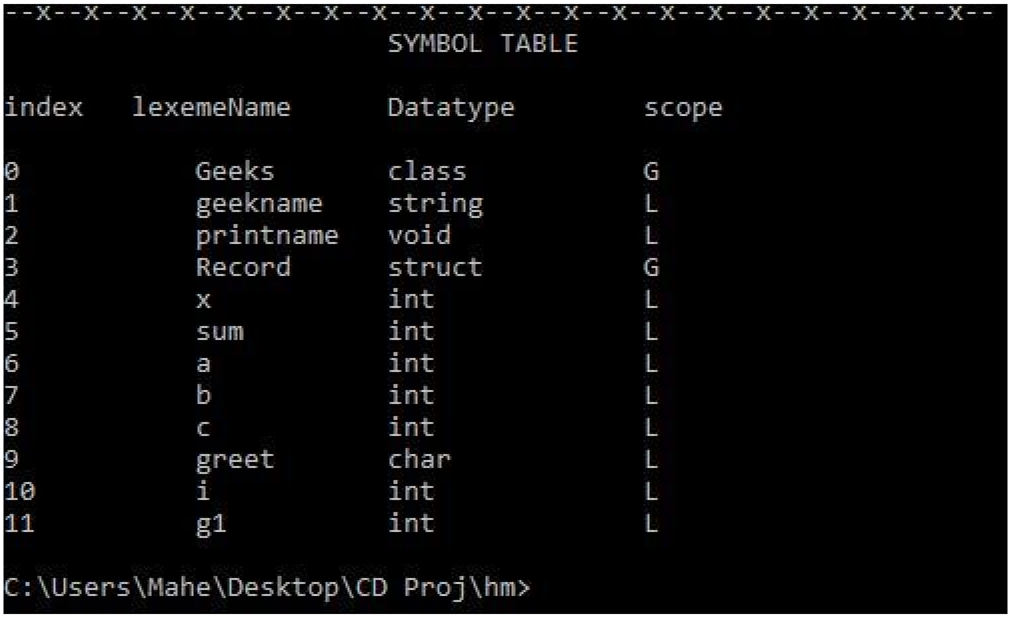
```
main() {
    int a,b,i;
    cin>>b;
    if(a<b) {
        a=a+1;
    } else if(a>b) {
        a=a-1;
    } else {
        a=2;
    }

    switch(a) {
        case 1:
            a=1;
            while(a!=2) {
                a=a-1;
            }
        case 2:
            a=3;
            for(i=0;i<10; i=i+1) {
                cout<<"Hello\n";
            }
            break;
    }
    g1.printname();
}
```



```
class Geeks
{
    // Access specifier
    public:

    // Data Members
    string geekname;

    // Member Functions()
    void printname()
    {
       cout << "Geekname is: " << geekname;
    }
};

struct Record {
    int x;
};

int sum (int a, int b) {
    int c;
    c = a+b;
    return c;
}

char greet() {
    char c;
    c=c+1;
    return c;
}
```



```
10
i
i
1
cout
"Hello\n"
break
g1
printname
SUCCESS
```



```
--X--X--X--X--X--X--X--X--X--X--X--X--X--X--X--X--X--X--X--X--
                        SYMBOL TABLE

index   lexemeName      Datatype        scope

0           Geeks       class           G
1           geekname    string          L
2           printname   void            L
3           Record      struct          G
4           x           int             L
5           sum         int             L
6           a           int             L
7           b           int             L
8           c           int             L
9           greet       char            L
10          i           int             L
11          g1          int             L

C:\Users\Mahe\Desktop\CD Proj\hm>
```