TITLE – COMPILER CONSTRUCTION (BCT 2311) ASSIGNMENT 2 C.T.3.2

18 JAN 2021

GROUP MEMBERS:

LEON KARIUKI - SCT212-7402/2015

SUSAN WAITHERA MWAURA - SCT212-0241/2017

MANDELA MITAU MUITHI - SCT212-8809/2015

DENNIS WAITHAKA - SCT212-5262/2015

LEWIS KATHEMEBE - SCT212-0339/2016

1. **Using the Recursive Descent strategy, write a C program for a simple calculator that can be used to perform integer arithmetic involving ‘+’ and ‘\*’. Let your program consist of a set of mutually recursive routines.**

/\*

valid := digits | '(' EXPR ')' )

calculate :=validOPvalid

EXPR :=OPdigits

OP := \* | +

digits := {+|-} [0..9] {[0..9]}

\*/

#include <stdio.h>

#include <ctype.h>

#include <stdlib.h>

char token;

int calculate(void);

//Method to report errors

void error(const char \*msg) {

fputs(msg, stderr);

exit(1);

}

// match expected and get next charcter

void match(char lexeme) {

if (token == lexeme) {

token = getchar();

return;

}

//print an errors

fprintf(stderr, "Expected %c, got %c", lexeme, token);

exit(1);

}

// validate each input character ( valid := digits | '(' EXPR ')' )

int validate(void) {

int value;

// example (2+3)

if (token == '(') {

match('(');

value = calculate();

match(')');

// isdigit(token) check for digits

} else if (isdigit(token) || token == '+' || token == '-') {

// return a character back onto an input stream.

ungetc(token, stdin);

scanf("%d", &value);

//gets next char.

token = getchar();

} else {

error("invalid input!");

}

return value;

}

// calculate :=digitsEXPR

int calculate(void) {

int value = validate();

while (token == '\*' || token == '+') {

switch(token) {

case '\*':

match('\*');

value =value \* validate();

break;

case '+':

match('+');

value =value + validate();

break;

default:

error("Wrong calculation!");

}

}

return value;

}

//main method

int main(void) {

token = getchar();

int result = calculate();

printf("result: %d\n", result);

return 1;

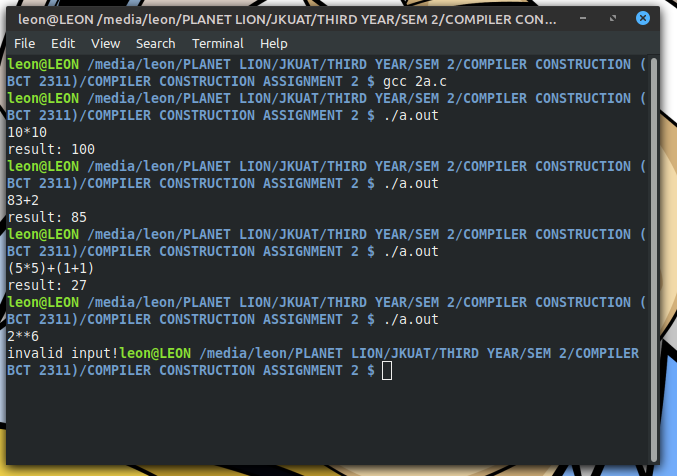
}

**1.gcc 2a.c gcc** - Command to compile the c program **(2a.c**)

**gcc** - Command to compile the c program (**2a.c**)

2. **./a.out**

**./** - command to execute the executable (**a.out**)



1. **Using the following grammar**

**S -> a S | b**

**Create an interpreter using LEX and YACC which will count the number of a’s in the input string.**

/\*Lex specification file\*/

%{

#include <stdio.h>

#include "y.tab.h"

%}

%%

"a" return \*yytext;

"b" return \*yytext;

[\n] return NEWLINE;

%%

int yywrap()

{

return 1;

}

/\*yacc file \*/

//declarations

%{

#include <stdio.h>

int count = 0;

int yylex();

extern int yyerror();

%}

%token NEWLINE

//grammar rules

%%

//identifier: definition

start : S NEWLINE { return 0; };

S: 'a' S { count++; }

| 'b' {}

;

%%

//functions

yyerror(char const \*s)

{

printf(" create a %s\n", s);

return 0;

}

int main()

{

printf("Enter the string\n");

yyparse();

printf("Number of a\'s: %d\n", count);

return 0;

}

**yacc -d 2b.y** **yacc** -command to compile the yacc file (**2b.y**).

**-d** optional flag which tells the yacc command to create a file that defines the tokens used in addition to the C language source code.

**flex 2b.l**

**flex** -command generate a scanner from the lex file (2b.l)

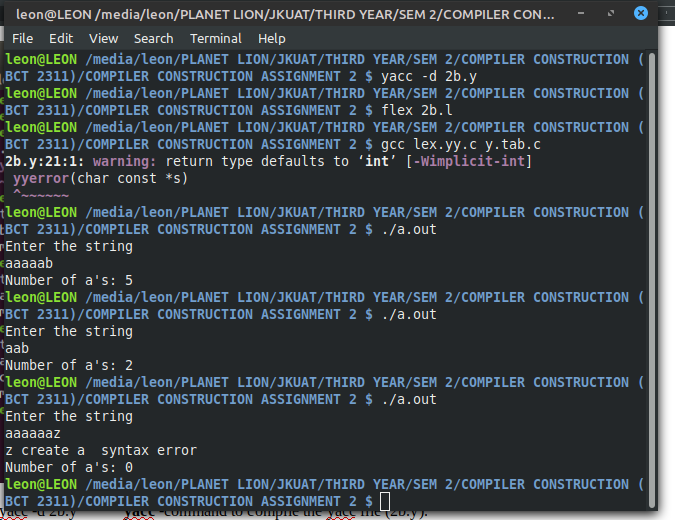
**gcc lex.yy.c y tab.c**

**gcc**- Compile and link the two C language source files:

**lex.yy.c**- The C language source file that the flex command created for the lexical analyzer.

**y.tab.c** -The C language source file that the yacc command created for the parser.

**./a.out ./** command torun the file( **a.out)**



1. **Write a LEX and YACC specification files for a small calculator that can add and subtract numbers.**

\*Lex File\*/

%{

#include <stdio.h>

#include "y.tab.h"

int c;

extern int yylval;

%}

%%

" " ;

[a-z] {

c = yytext[0];

yylval = c - 'a';

return(LETTER);

}

[0-9] {

c = yytext[0];

yylval = c - '0';

return(DIGIT);

}

[^a-z0-9\b] {

c = yytext[0];

return(c);

}

/\*yacc file for a simple calculator\*/

%{

#include <stdio.h>

int regs[26];

int base;

int yyerror();

%}

%start list

%token DIGIT LETTER

%left '|'

%left '+' '-'

%left UMINUS //supplies precedence for unary minus

%% //beginning of rules section

list: //empty

|

list stat '\n'

|

list error '\n'

{

yyerrok;

}

;

stat: expr

{

printf("%d\n",$1);

}

|

LETTER '=' expr

{

regs[$1] = $3;

}

;

expr: '(' expr ')'

{

$$ = $2;

}

|

expr '+' expr

{

$$ = $1 + $3;

}

|

expr '-' expr

{

$$ = $1 - $3;

}

|

'-' expr %prec UMINUS

{

$$ = -$2;

}

|

LETTER

{

$$ = regs[$1];

}

number

;

number: DIGIT

{

$$ = $1;

base = ($1==0) ? 8 : 10;

} |

number DIGIT

{

$$ = base \* $1 + $2;

}

;

%%

int main()

{

return(yyparse());

}

int yyerror(s)

char \*s;

{

fprintf(stderr, "%s\n",s);

}

int yywrap()

{

return(1);

}

