**Compilers Lab**

**COE – 406**

By – Divjot Singh

262/CO/12

COE – 1

**Index**

1. Write a two-pass assembler for subset of 8085 instruction set.
2. Write context free grammar for a subset of C Programming language.
3. Create basic programs using lex.
4. Create a BODMAS based calculator in yacc with variables.

**1) Assembler for 8085 [JavaScript, NodeJS]**

**index.js**

'use strict';

var asm = {};

const iSet = require('./instructionSet');

/\* Assembles the given code into machine code \*/

asm.assemble = (code, showAddress) => {

showAddress = showAddress || false;

let labels = {};

let startWith = 0;

let assembledCode = [];

/\* Preprocessing : Decomment \*/

let codeLines = asm.decomment(code).split('\n').map(cl => cl.trim());

/\* First Pass: Get all labels in labels obj \*/

codeLines.reduce((startWith, codeLine) => {

if(asm.isORGDirective(codeLine)) {

startWith = asm.getORGLocation(codeLine);

} else {

if(asm.isLabel(codeLine)) {

labels[asm.getLabel(codeLine)] = asm.toHex(startWith);

}

startWith += asm.getInstructionSize(codeLine);

}

return startWith;

}, startWith);

// console.log(labels);

/\* Second Pass: Expand mnemonics, replace label tags with actual line number \*/

startWith = -1;

codeLines.forEach((codeLine, index) => {

if(asm.isORGDirective(codeLine)) {

startWith = asm.getORGLocation(codeLine) - 1;

} else {

let iFormat = asm.sanitize(codeLine);

let operands = asm.getInstructionOperands(codeLine);

let lastOperand = operands[operands.length - 1];

/\* First step for each instruction \*/

assembledCode.push({ address: asm.toHex(++startWith), code: iSet[iFormat].code });

switch(asm.getInstructionSize(codeLine)) {

case 2: assembledCode.push({ address: asm.toHex(++startWith), code: lastOperand }); break;

case 3:

lastOperand = (asm.isLabelInstruction(codeLine) && lastOperand in labels) ? labels[lastOperand] : lastOperand;

assembledCode.push({ address: asm.toHex(++startWith), code: lastOperand.slice(-2) });

assembledCode.push({ address: asm.toHex(++startWith), code: lastOperand.slice(-4, -2) });

break;

}

}

});

return assembledCode.reduce((string, value) => string += ((showAddress? (value.address + ' ') : '') + value.code + '\n'), '');

};

/\* Returns the size (in bytes) of the instruction \*/

asm.getInstructionSize = codeLine => {

if(!asm.isValidInstruction(codeLine)) {

throw new Error('Instruction "' + codeLine + '" doesn\'t match to any instruction');

}

return iSet[asm.sanitize(codeLine)].size;

};

/\* Converts a number to 'data' or 'address' tag which is used in formal instruction format \*/

asm.numberToTag = codeLine => {

let operands = asm.getInstructionOperands(codeLine);

if(operands.length > 0) {

if(asm.isDataInstruction(codeLine)) {

codeLine = codeLine.replace(operands[operands.length - 1], 'data');

} else if(asm.isLabelInstruction(codeLine)) {

codeLine = codeLine.replace(operands[operands.length - 1], 'address');

}

}

return codeLine;

};

/\* Remove comments, anything starting with ; \*/

asm.decomment = code => code.replace(/(;.\*)/g, '').trim();

/\* Tells whether given line has a label to it \*/

asm.isLabel = codeLine => codeLine.split(' ')[0].endsWith(':');

/\* Returns label of a code line with a label to it \*/

asm.getLabel = codeLine => codeLine.split(' ')[0].slice(0, -1);

/\* Checks whether given code line is a DB directive \*/

asm.isDBDirective = codeLine => codeLine.trim().startsWith('# DB');

/\* Checks whether given code line is an ORG directive \*/

asm.isORGDirective = codeLine => codeLine.trim().startsWith('# ORG');

/\* Checks whether given code line is an DB directive \*/

asm.isDBDirective = codeLine => codeLine.trim().startsWith('# DB');

/\* Checks validity of code line based on instruction set\*/

asm.isValidInstruction = codeLine => asm.sanitize(codeLine) in iSet;

/\* Removes label from a given code line, if any \*/

asm.removeLabel = codeLine => codeLine.replace(/[a-zA-Z]+:/g, '').trim();

/\* Returns the decimal location where an ORG directive instruction points to \*/

asm.getORGLocation = orgLine => parseInt(orgLine.replace('# ORG', '').trim(), 16);

/\* Returns array of data to be stored as per DB Directive \*/

asm.getDBOperands = dbLine => dbLine.replace('# DB', '').split(',').map(e => e.trim());

/\* Returns the mnemonic of main instruction \*/

asm.getInstructionName = codeLine => codeLine.split(' ')[(asm.isLabel(codeLine) ? 1 : 0)];

/\* Returns upper case hex string of 4 digits for given decimal number \*/

asm.toHex = decimalNumber => ("000" + decimalNumber.toString(16).toUpperCase()).slice(-4);

/\* Converts a code line to formal instruction format as per Instruction Set \*/

asm.sanitize = codeLine => asm.labelToTag(asm.numberToTag(asm.removeLabel(codeLine))).trim();

/\* Checks whether given instruction belongs to ones which have immediate data operand \*/

asm.isDataInstruction = codeLine => iSet.dataInstructions.indexOf(asm.getInstructionName(codeLine)) > -1;

/\* Checks whether given instruction belongs to ones which have an address/label operand \*/

asm.isLabelInstruction = codeLine => iSet.labelInstructions.indexOf(asm.getInstructionName(codeLine)) > -1;

/\* Converts a label to 'address' tag which is used in formal instruction format.

\* TODO: labels can also point to data, say by using EQU \*/

asm.labelToTag = codeLine => asm.isLabelInstruction(codeLine) ? codeLine.replace(codeLine.split(' ').slice(-1), 'address') : codeLine;

/\* Returns array of operands of an instruction \*/

asm.getInstructionOperands = codeLine => asm.removeLabel(codeLine).replace(asm.getInstructionName(codeLine), '').split(',').map(e => e.trim());

module.exports = asm;

**instructionSet.js (Contains entire instruction set of 8085 and metadata)**

'use strict';

module.exports = {

dataInstructions: ['ACI', 'ADI', 'ANI', 'MVI', 'CPI', 'ORI', 'SBI', 'SUI', 'XRI', 'LXI'],

labelInstructions: ['JMP', 'JC', 'JM', 'JNC', 'JNZ', 'JP', 'JPE', 'JPO', 'JZ', 'CALL', 'CC', 'CM', 'CNC', 'CNZ', 'CP', 'CPE', 'CPO', 'CZ', 'LDA', 'LHLD', 'SHLD', 'STA'],

'ACI data': {

'size': 2,

'code': 'CE',

'desc': 'Add with Carry Immediate'

},

'ADC A': {

'size': 1,

'code': '8F',

'desc': 'Add with Carry'

},

…

'XTHL': {

'size': 1,

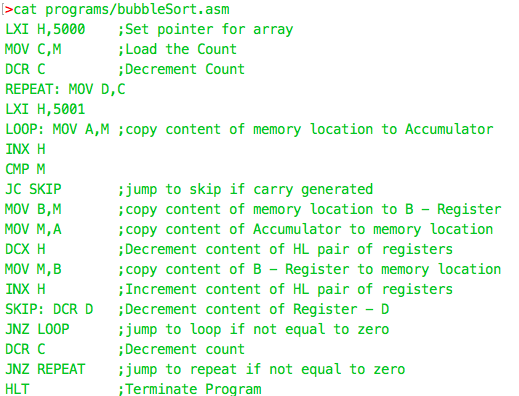
'code': 'E3',

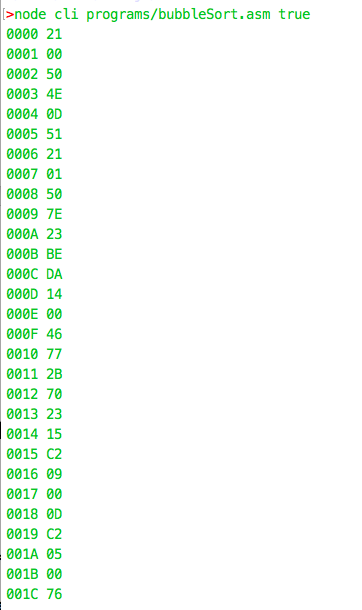
'desc': 'Exchange stack Top with HL'

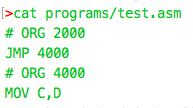
}

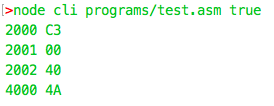
};

**Output**

****

****

****

****

**2) Context Free Grammar for subset of C**

**Start**

<start> -> <pre-processor-directives> <function-declarations> <main-function> FUNCTION

<pre-processor-directives> -> #include<L> | #include"S" | #define <identifier> M| typdef <type><identifier>;

L -> A-Za-z

M -> [0-9]+ | ".\*"

S -> \S\*

<main-function> -> int main(Z)S

Z -> void | intI, char\*I[] | int I,char\*\*

<function-declarations> -> FUNCTIONDEC

**Types**

<types> -> P | P\*

P -> int | char | float | double | void

**Identifier**

<identifier> -> [a-zA-Z\_$][a-zA-Z\_$0-9]\*

**Expressions**

**E - Expression**

<expression> -> <identifier> | <number> |E+F | E-F

F -> F\*G | F/G

G -> (E)

**Declaration Expressions**

<declaration> -> <type> <identifier>

**Assigment Expression**

<assignment> -> <identifier>=<expression> | <declaration> = <expression>

**Relational Expression**

<relational> -> <expression> < <expression>

| <expression> > <expression>

| <expression> <= <expression>

| <expression> != <expression>

| <expression> == <expression>

| <expression> && <expression>

| <expression> || <expression>

| ! <relational>

**Statements**

**Statement**

<statement> -> ; | <expression>;| {<statement>} | L | J | IF | SWITCH | FOR | WHILE | DO | CALL

**C - Case Statements**

C -> case<identifier>:<statement> | default:<statement> | C

**L - Labelled Statements**

L -> <identifier>:<statement>

**J - Jump Statements**

J -> return; | return<expression>; | goto<identifier>; | break; | continue;

**Branching Constructs**

**if construct**

IF -> if(<expression>)<statement>else<statement> | if(<expression>)<statement>

**switch construct**

SWITCH -> switch(<expression>){C}

**Loop Constructs**

**for loop construct**

FOR -> for(;;)S | for(<expression>;<relational>;)<statement> | for(<expression>;<relational>;<expression>)<statement>

**while loop construct**

WHILE -> while(<relational>)<statement>

**do while loop construct**

DO -> do{<statement>}while(<relational>);

**Functions**

**Function Declaration**

FUNCTIONDEC -> <declaration> (P);

P -> <declaration> | <declaration>, <declaration>

**Function Definition**

FUNCTION -> <declaration> (P)<statement> | <identifier>(P)<statement>

**Function Call**

CALL -> <identifier>(Z);

Z -> <identifier> | <identifier>, <identifier>

**3) Lex Programs**

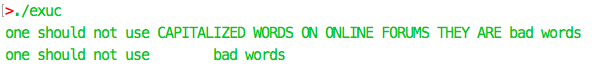
**Remove Upper Case Letters**

**Code**

%%

[A-Z]+

**Output**

****

**Line Numbering**

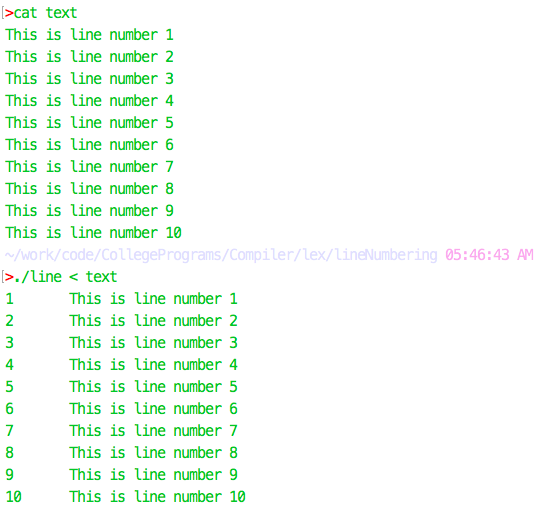
**Code**

%%

\n { yylineno++; printf("\n"); };

^.\*$ printf("%d\t%s", yylineno, yytext);

**Output**

****

**Word Count**

**Code**

%{

int charCount = 0, wordCount = 0, lineCount = 0;

%}

%%

\n ++charCount, ++lineCount;

[^ \t\n]+ ++wordCount, charCount += yyleng;

. ++charCount;

%%

int main (void) {

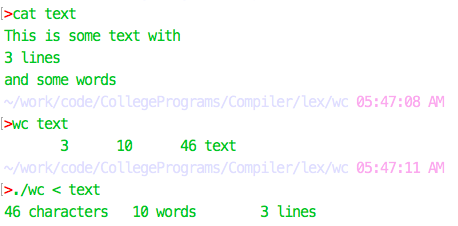
yylex();

printf("%d characters\t%d words\t%d lines\n", charCount, wordCount, lineCount);

return 0;

}

**Output**

****

**File Inclusion**

**Code**

%{

#include <ctype.h>

#include <stdio.h>

static void include (char\* s);

static char\* trim (char\* s);

%}

%%

^"#include".\*\n { yytext[yyleng - 1] = '\0'; include(yytext + 8);}

%%

static char\* trim (char\* s) {

while (\*s && isspace(\*s)) s++;

return s;

}

static void include (char\* s) {

FILE\* fp;

int i;

char\* fileName = trim(s);

if ((fp = fopen(fileName, "r"))) {

while ((i = getc(fp)) != EOF) printf("%c", i);

fclose(fp);

} else {

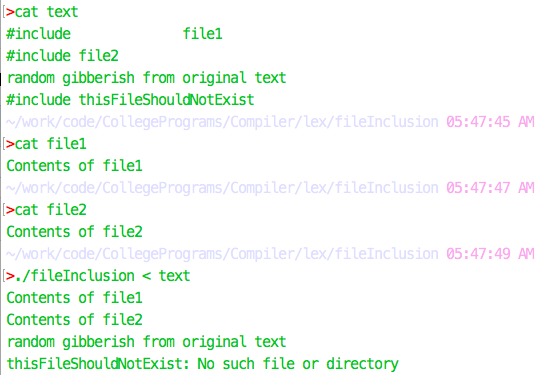
perror(fileName);

}

return;

}

**Output**

****

**Lexical Analyzer for C**

**Code**

%{

/\* need this for the call to atof() below \*/

#include <math.h>

%}

WHITESPACE [ \t]\*

TYPE "int"|"char"|"float"|"double"|"void"

OPERATOR "+"|"-"|"<"|">"|"\*"|"/"|"="

DIGIT [0-9]

LETTER [a-zA-Z]

NUMBER {DIGIT}+(\.{DIGIT}+)?(e[+\-]?{DIGIT}+)?

IDENTIFIER [a-zA-Z\_$][a-zA-Z\_$0-9]\*

JUMP break|return|continue|goto

%%

{TYPE} printf("A data type: %s\n", yytext);

{DIGIT}+ printf("An integer: %s (%d)\n", yytext, atoi(yytext));

{DIGIT}+"."{DIGIT}\* printf("A floating constant: %s (%g)\n", yytext, atof(yytext));

{JUMP} printf("An jump statement: %s\n",yytext );

{IDENTIFIER} printf("An identifier: %s\n", yytext);

{OPERATOR} printf("An operator: %s\n",yytext );

%%

int main(void) {

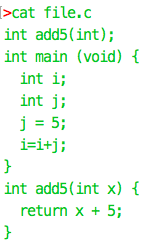
yyin = stdin;

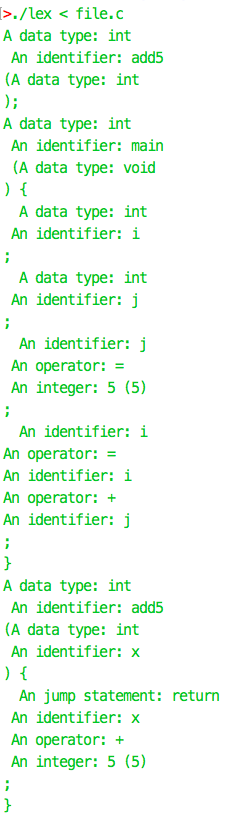
yylex();

return 0;

}

**Output**

****

****

**4) Calculator in yacc**

**calc.y**

%{

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void yyerror (char \*s);

int symbols[52];

int getSymbolValue (char symbol);

void setSymbolValue (char symbol, int val);

%}

%union {int num; char id;}

%start line

// Some #defines

%token print

%token exit\_command

// Some #defines with their types

%token <num> number

%token <id> identifier

%type <num> line exp exp1 exp2 term

%type <id> assignment

%%

/\* descriptions of expected inputs corresponding actions (in C) \*/

line: assignment ';' { ; }

| exit\_command ';' { exit(EXIT\_SUCCESS); }

| print exp ';' { printf("Printing %d\n", $2); }

| line assignment ';' { ; }

| line print exp ';' { printf("Printing %d\n", $3); }

| line exit\_command ';' { exit(EXIT\_SUCCESS); }

;

assignment : identifier '=' exp { setSymbolValue($1,$3); }

;

exp : term { $$ = $1; }

| exp '+' exp1 { $$ = $1 + $3; }

| exp '-' exp1 { $$ = $1 - $3; }

| exp1 { $$ = $1; }

;

exp1 : term { $$ = $1; }

| exp1 '\*' exp2 { $$ = $1 \* $3; }

| exp1 '/' exp2 { $$ = $1 / $3; }

| exp2 { $$ = $1; }

;

exp2 : term { $$ = $1; }

| '(' exp ')' { $$ = $2; }

;

term : number { $$ = $1; }

| identifier { $$ = getSymbolValue($1); }

;

%%

int computeSymbolIndex (char token) {

int idx = -1;

if(islower(token)) {

idx = token - 'a' + 26;

} else if(isupper(token)) {

idx = token - 'A';

}

return idx;

}

int getSymbolValue(char symbol) {

return symbols[computeSymbolIndex(symbol)];

}

void setSymbolValue(char symbol, int val) {

symbols[computeSymbolIndex(symbol)] = val;

}

int main (void) {

int i;

// Initialize symbol table

for(i=0; i < 52; i++) {

symbols[i] = 0;

}

return yyparse ( );

}

void yyerror (char \*s) {

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

}

**calc.l**

%{

#include "y.tab.h"

extern char\* yytext;

extern void yyerror (char \*s);

%}

%%

"print" { return print; }

"exit" { return exit\_command; }

[a-zA-Z] { yylval.id = yytext[0]; return identifier; }

[0-9]+ { yylval.num = atoi(yytext); return number; }

[ \t\n] { ; }

[-+/\*()=;] { return yytext[0]; }

. { printf("%s", yytext); yyerror ("Unexpected character"); }

%%

int yywrap (void) { return 1; }

**Output**

