**COVER PAGE**

# **CS323 Programming Assignments**

**Fill out all entries 1 - 7. If not, there will be deductions!**

**Check one**

1. Names [ 1. Liren Yin CPSC323-02 ], (4pm class [ ] or 5:30pm class [ X ] )

[ 2.Yafei Mo CPSC323-04], (4pm class [ ] or 5:30pm class [ X ] )

2. Assignment Number [ 3 ]

3. Due Dates **Softcopy**  [ Monday 5/7/2018 11:59PM]

**Hardcopy** [Tuesday 5/8/2018 8:15PM]

4. Turn-In Dates **Softcopy** [ Thursday 5/3/2018 10:00AM ], **Hardcopy** [Tuesday 5/8/2018 5:15PM ]

5. Executable FileName [ CPSC323\_HW3 ]

(**A file that can be executed without compilation by the instructor**)

6. LabRoom [ CS101 ]

**(Execute your program in a lab in the CS building before submission)**

7. Operating System [ MacOS Sierra 10.12.6 ]

**To be filled out by the Instructor:**

GRADE:

COMMENTS:

1. **Problem Statement**

*We are building a customize compiler Rat18S with our own defined programming language. At this stage, we are going to design a semantic analyzer for this compiler. Output of the program consists of symbol table and assembly code.*

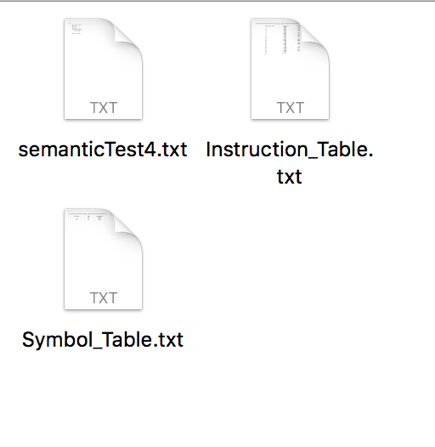
1. **How to use your program**
2. *The environment for running this program will be macOS High Sierra (V10.12.6).*
3. *Open the folder* ***CPSC323\_HW3-cbkgicrkfjoyzzbxhprccbojmcoy/Build/Products/Debug***
4. *Open file* ***CPSC323\_HW3***

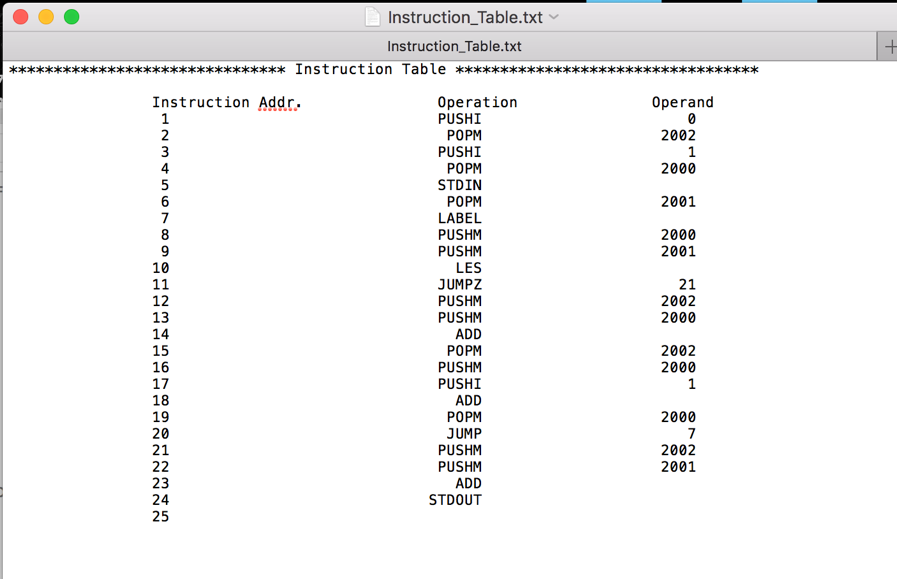
*A console will pop up, and there’s a default file path on the screen*

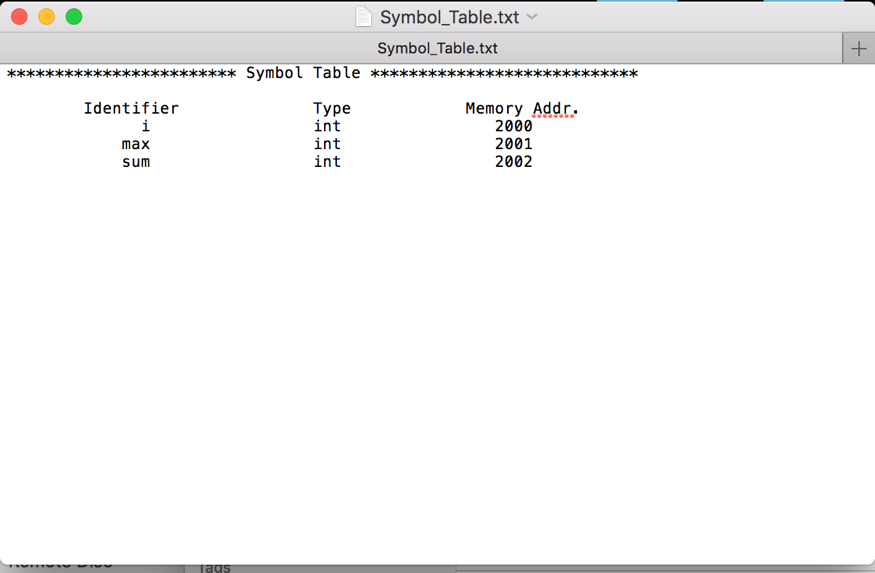
**

*In this case, the default file path is /Users/liren*

1. *Copy the source code file expected to be dealt with.*
2. *Go to default path (/Users/liren in this case), paste the source code you would like to parse here.*
3. *In the console, type in the file name you would like to parse (Ex. SemanticTest4.txt)*
4. *The outputs of lexical analysis, syntax analysis, list of identifiers, symbol table and instruction table will show on the console screen. In addition, the program will generate two files. “Instruction\_Table.txt” contains assembly codes and “Symbol\_Table.txt” contains all identifiers with corresponding types and memory addresses.*

**





1. **Design of your program**

Predictive recursive descent parser is used in this homework design, at the mean time of producing parsing tree, assembly code is produced during the recursive functions are called when certain production rules are matched.

1. **Any Limitation**

*None*

1. **Any shortcomings**

*It’s not working under Microsoft Windows Enviornment.*

**Test Cases:**

*semanticTest1.txt*

%%

int x, a, c, d, e, b, i, max, low, high, step;

while (i < max) i = i + 1;

x = a + b \* c/d-e;

if (a < b) a = c;

else a = b + c;

endif

get(low, high, step);

a = put(low - high \* step);

c = a \* d - e;

while (i < 10) {

a = c + 1;

i = i + 1;

}

Instruction\_Table.txt

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Symbol Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Identifier Type Memory Addr.**

**x int 2000**

**a int 2001**

**c int 2002**

**d int 2003**

**e int 2004**

**b int 2005**

**i int 2006**

**max int 2007**

**low int 2008**

Symbol\_Table.txt

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Instruction Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Instruction Addr. Operation Operand**

**1 LABEL**

**2 PUSHM 2006**

**3 PUSHM 2007**

**4 LES**

**5 JUMPZ 11**

**6 PUSHM 2006**

**7 PUSHI 1**

**8 ADD**

**9 POPM 2006**

**10 JUMP 1**

**11 PUSHM 2001**

**12 PUSHM 2005**

**13 PUSHM 2002**

**14 MUL**

**15 PUSHM 2003**

**16 DIV**

**17 ADD**

**18 PUSHM 2004**

**19 SUB**

**20 POPM 2000**

**21 PUSHM 2001**

**22 PUSHM 2005**

**23 LES**

**24 JUMPZ 27**

**25 PUSHM 2002**

**26 POPM 2001**

**27 PUSHM 2005**

**28 PUSHM 2002**

**29 ADD**

**30 POPM 2001**

**31 STDIN**

**32 POPM 2008**

**33 STDIN**

**34 POPM 2009**

**35 STDIN**

**36 POPM 2010**

**37 PUSHM 2008**

**38 PUSHM 2009**

**39 PUSHM 2010**

**40 MUL**

**41 SUB**

**42 STDOUT**

**43 PUSHM 2001**

**44 PUSHM 2003**

**45 MUL**

**46 PUSHM 2004**

**47 SUB**

**48 POPM 2002**

**49 LABEL**

**50 PUSHM 2006**

**51 PUSHI 10**

**52 LES**

**53 JUMPZ 63**

**54 PUSHM 2002**

**55 PUSHI 1**

**56 ADD**

**57 POPM 2001**

**58 PUSHM 2006**

**59 PUSHI 1**

**60 ADD**

**61 POPM 2006**

**62 JUMP 49**

**63**

*semanticTest2.txt*

%%

int a, b, c;

if (a < b) a = c; endif

Instruction\_Table.txt

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Symbol Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Identifier Type Memory Addr.**

**a int 2000**

**b int 2001**

**c int 2002**

Symbol\_Table.txt

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Instruction Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Instruction Addr. Operation Operand**

**1 PUSHM 2000**

**2 PUSHM 2001**

**3 LES**

**4 JUMPZ 7**

**5 PUSHM 2002**

**6 POPM 2000**

**7**

*semanticTest3.txt*

%%

int i, max, sum;

boolean hello;

sum = 0;

i = 1;

get (max);

while (i < max) {

sum = sum + i;

i = i + 1;

}

put (sum + max);

hello = true;

if (hello == true) {

sum = sum + 3;

}

endif

if (hello == false) {

sum = 0;

max = 0;

}

endif

Instruction\_Table.txt

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Symbol Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Identifier Type Memory Addr.**

**i boolean 2000**

**max boolean 2001**

**sum boolean 2002**

**hello boolean 2003**

Symbol\_Table.txt

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Instruction Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Instruction Addr. Operation Operand**

**1 PUSHI 0**

**2 POPM 2002**

**3 PUSHI 1**

**4 POPM 2000**

**5 STDIN**

**6 POPM 2001**

**7 LABEL**

**8 PUSHM 2000**

**9 PUSHM 2001**

**10 LES**

**11 JUMPZ 21**

**12 PUSHM 2002**

**13 PUSHM 2000**

**14 ADD**

**15 POPM 2002**

**16 PUSHM 2000**

**17 PUSHI 1**

**18 ADD**

**19 POPM 2000**

**20 JUMP 7**

**21 PUSHM 2002**

**22 PUSHM 2001**

**23 ADD**

**24 STDOUT**

**25 POPM 2003**

**26 PUSHM 2003**

**27 EQU**

**28 JMPZ 33**

**29 PUSHM 2002**

**30 PUSHI 3**

**31 ADD**

**32 POPM 2002**

**33 PUSHM 2003**

**34 EQU**

**35 JMPZ 40**

**36 PUSHI 0**

**37 POPM 2002**

**38 PUSHI 0**

**39 POPM 2001**

**40**

**Source Code:**

//

// main.cpp

// CPSC323\_HW3

//

// Created by Liren on 4/20/18.

// Copyright © 2018 Liren. All rights reserved.

//

//main.cpp

#include <iostream>

#include <fstream>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <iomanip>

#include "LexAnalysis.h"

#include "SynAnalysis.h"

int main()

{

bool tokenError = false;

system("pwd");

NormalNode\* SyntaxPartHead;

NormalNode\* SemanticPartHead;

//Lexical Analysis Part

initKeyMapping();

initOperMapping();

initLimitMapping();

initNode();

scanner();

BraMappingError();

SyntaxPartHead = printNodeLink();

SemanticPartHead = SyntaxPartHead;

// printErrorLink(tokenError);

// if (tokenError) {

// std::cout << "Since Token Error exists, Syntax Analysis will not proceed.\n";

// return 0;

// }

printIdentLink();

//Syntax Analysis Part

analysis(\*SyntaxPartHead);

}

//

// LexAnalysis.h

#ifndef LexAnalysis\_h

#define LexAnalysis\_h

//Keywords

#define AUTO 1

#define BREAK 2

#define CASE 3

#define CHAR 4

#define CONST 5

#define CONTINUE 6

#define DEFAULT 7

#define DO 8

#define DOUBLE 9

#define ELSE 10

#define FLOAT 13

#define FOR 14

#define GOTO 15

#define IF 16

#define INT 17

#define LONG 18

#define REGISTER 19

#define RETURN 20

#define SHORT 21

#define SIGNED 22

#define SIZEOF 23

#define STATIC 24

#define STRUCT 25

#define SWITCH 26

#define TYPEDEF 27

#define UNION 28

#define UNSIGNED 29

#define VOID 30

#define VOLATILE 31

#define WHILE 32

#define REAL 33

#define PUT 34

#define GET 35

#define FUNCTION 36

#define BOOLEAN 37

#define ENDIF 38

#define TRUE\_VALUE 11

#define FALSE\_VALUE 12

#define KEY\_DESC "Keyword"

//Identifiers

#define IDENTIFER 40

#define IDENTIFER\_DESC "Identifier"

//Constant

#define INT\_VAL 51 //int constant

#define MACRO\_VAL 55 //macro constant

#define REAL\_VAL 56 // Real Numbers

#define REAL\_DESC "real"

#define INT\_DESC "int"

//Operators

#define MUL 65 // \*

#define DIV 66// /

#define ADD 68 // +

#define SUB 69 // -

#define LES\_THAN 70 // <

#define GRT\_THAN 71 // >

#define ASG 72 // =

#define SELF\_ADD 74 // ++

#define SELF\_SUB 75 // --

#define LEFT\_MOVE 76 // <<

#define RIGHT\_MOVE 77 // >>

#define LES\_EQUAL 78 // =>

#define GRT\_EQUAL 79 // =<

#define EQUAL 80 // ==

#define COMPLETE\_BYTE\_XOR 88 // ^=

#define PERCENTPERCENT 91 //%%

#define BYTE\_OR 92 // |

#define OPE\_DESC "Operator"

//Seperators

#define LEFT\_BRA 100 // (

#define RIGHT\_BRA 101 // )

#define LEFT\_INDEX 102 // [

#define RIGHT\_INDEX 103 // ]

#define L\_BOUNDER 104 // {

#define R\_BOUNDER 105 // }

#define POINTER 106 // .

#define JING 107 // #

#define UNDER\_LINE 108 // \_

#define COMMA 109 // ,

#define SEMI 110 // ;

#define COLON 81 // :

#define SIN\_QUE 111 // '

#define DOU\_QUE 112 // "

#define CLE\_OPE\_DESC "Seperator"

#define NOTE1 120 // "!!"comment

#define NOTE\_DESC "comment"

#define HEADER 130 //header

#define HEADER\_DESC "header"

//Error Types

#define REAL\_ERROR "real type error"

#define REAL\_ERROR\_NUM 1

#define DOUBLE\_ERROR "double type error"

#define DOUBLE\_ERROR\_NUM 2

#define NOTE\_ERROR "comment format error"

#define NOTE\_ERROR\_NUM 3

#define STRING\_ERROR "String constant error"

#define STRING\_ERROR\_NUM 4

#define CHARCONST\_ERROR "Char constant error"

#define CHARCONST\_ERROR\_NUM 5

#define CHAR\_ERROR "Invalid Char"

#define CHAR\_ERROR\_NUM 6

#define LEFT\_BRA\_ERROR "'('not matching"

#define LEFT\_BRA\_ERROR\_NUM 7

#define RIGHT\_BRA\_ERROR "')'not matching"

#define RIGHT\_BRA\_ERROR\_NUM 8

#define LEFT\_INDEX\_ERROR "'['not matching"

#define LEFT\_INDEX\_ERROR\_NUM 9

#define RIGHT\_INDEX\_ERROR "']'not matching"

#define RIGHT\_INDEX\_ERROR\_NUM 10

#define L\_BOUNDER\_ERROR "'{'not matching"

#define L\_BOUNDER\_ERROR\_NUM 11

#define R\_BOUNDER\_ERROR "'}'not matching"

#define R\_BOUNDER\_ERROR\_NUM 12

#define IDENTIFIER\_ERROR "Identifier Error" //Invalid identifier

#define IDENTIFIER\_ERROR\_NUM 13

#define \_NULL "null"

#define DESCRIBE 4000

#define TYPE 4001

#define STRING 4002

#define DIGIT 4003

struct NormalNode

{

char content[30];

char describe[30];

int type;

int addr;

int line;

NormalNode \* next;

NormalNode \* prev;

};

void initKeyMapping();

void initOperMapping();

void initLimitMapping();

void initNode();

void createNewNode(char \* content,char \*descirbe,int type,int addr,int line);

void createNewError(char \* content,char \*descirbe,int type,int line);

int createNewIden(char \* content,char \*descirbe,int type,int addr,int line);

NormalNode\* printNodeLink();

void printErrorLink(bool &TokenError);

void printIdentLink();

int mystrlen(char \* word);

void preProcess(char \* word,int line);

void close();

int seekKey(char \* word);

void scanner();

void BraMappingError();

#endif /\* LexAnalysis\_h \*/

//LexAnalysis.cpp

#include <iostream>

#include <fstream>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <vector>

#include <iomanip>

#include "LexAnalysis.h"

using namespace std;

int leftSmall = 0;//(

int rightSmall = 0;//)

int leftMiddle = 0;//[

int rightMiddle = 0;//]

int leftBig = 0;//{

int rightBig = 0;//}

int lineBra[6][1000] = {0};

int static\_iden\_number = 0;

NormalNode \* normalHead;

NormalNode \* normalTail;

struct ErrorNode

{

char content[30];

char describe[30];

int type;

int line;

ErrorNode \* next;

};

ErrorNode \* errorHead;

struct IdentiferNode

{

char content[30];

char describe[30];

int type;

int addr;

int line;

IdentiferNode \* next;

IdentiferNode \* prev;

};

IdentiferNode \* idenHead;

IdentiferNode \* idenTail;

vector<pair<const char \*,int> > keyMap;

vector<pair<const char \*,int> > operMap;

vector<pair<const char \*,int> > limitMap;

void initKeyMapping()

{

keyMap.clear();

keyMap.push\_back(make\_pair("else",ELSE));

keyMap.push\_back(make\_pair("true",TRUE\_VALUE));

keyMap.push\_back(make\_pair("false",FALSE\_VALUE));

keyMap.push\_back(make\_pair("if",IF));

keyMap.push\_back(make\_pair("int",INT));

keyMap.push\_back(make\_pair("real",REAL));

keyMap.push\_back(make\_pair("long",LONG));

keyMap.push\_back(make\_pair("register",REGISTER));

keyMap.push\_back(make\_pair("return",RETURN));

keyMap.push\_back(make\_pair("while",WHILE));

keyMap.push\_back(make\_pair("put",PUT));

keyMap.push\_back(make\_pair("get",GET));

keyMap.push\_back(make\_pair("function",FUNCTION));

keyMap.push\_back(make\_pair("boolean",BOOLEAN));

keyMap.push\_back(make\_pair("endif",ENDIF));

keyMap.push\_back(make\_pair("id",IDENTIFER));

}

void initOperMapping()

{

operMap.clear();

operMap.push\_back(make\_pair("\*",MUL));

operMap.push\_back(make\_pair("/",DIV));

operMap.push\_back(make\_pair("+",ADD));

operMap.push\_back(make\_pair("-",SUB));

operMap.push\_back(make\_pair("<",LES\_THAN));

operMap.push\_back(make\_pair(">",GRT\_THAN));

operMap.push\_back(make\_pair("=",ASG));

operMap.push\_back(make\_pair("=<",LES\_EQUAL));

operMap.push\_back(make\_pair("=>",GRT\_EQUAL));

operMap.push\_back(make\_pair("==",EQUAL));

operMap.push\_back(make\_pair(":",COLON));

operMap.push\_back(make\_pair("||",OR));

operMap.push\_back(make\_pair("^=",COMPLETE\_BYTE\_XOR));

operMap.push\_back(make\_pair("%%",PERCENTPERCENT));

}

void initLimitMapping()

{

limitMap.clear();

limitMap.push\_back(make\_pair("(",LEFT\_BRA));

limitMap.push\_back(make\_pair(")",RIGHT\_BRA));

limitMap.push\_back(make\_pair("[",LEFT\_INDEX));

limitMap.push\_back(make\_pair("]",RIGHT\_INDEX));

limitMap.push\_back(make\_pair("{",L\_BOUNDER));

limitMap.push\_back(make\_pair("}",R\_BOUNDER));

limitMap.push\_back(make\_pair(".",POINTER));

limitMap.push\_back(make\_pair(",",COMMA));

limitMap.push\_back(make\_pair(";",SEMI));

limitMap.push\_back(make\_pair("'",SIN\_QUE));

}

void initNode()

{

normalHead = new NormalNode();

normalTail = new NormalNode();

strcpy(normalHead->content,"");

strcpy(normalHead->describe,"");

normalHead->type = -1;

normalHead->addr = -1;

normalHead->line = -1;

normalHead->next = NULL;

normalHead->prev = NULL;

normalTail->type = -1;

normalTail->addr = -1;

normalTail->line = -1;

normalTail->next = NULL;

normalTail->prev = NULL;

errorHead = new ErrorNode();

strcpy(errorHead->content,"");

strcpy(errorHead->describe,"");

errorHead->line = -1;

errorHead->next = NULL;

idenHead = new IdentiferNode();

idenTail = new IdentiferNode();

strcpy(idenHead->content,"");

strcpy(idenHead->describe,"");

idenHead->type = -1;

idenHead->addr = -1;

idenHead->line = -1;

idenHead->next = NULL;

idenHead->prev = NULL;

idenTail->type = -1;

idenTail->addr = -1;

idenTail->line = -1;

idenTail->next = NULL;

idenTail->prev = NULL;

}

void createNewNode(char \* content,char \*descirbe,int type,int addr,int line)

{

NormalNode \* p = normalHead;

NormalNode \* temp = new NormalNode();

while(p->next != NULL)

{

p = p->next;

}

strcpy(temp->content,content);

strcpy(temp->describe,descirbe);

temp->type = type;

temp->addr = addr;

temp->line = line;

temp->next = NULL;

p->next = temp;

temp->prev = p;

normalTail = temp;

}

void createNewError(char \* content,char \*descirbe,int type,int line)

{

ErrorNode \* p = errorHead;

ErrorNode \* temp = new ErrorNode();

strcpy(temp->content,content);

strcpy(temp->describe,descirbe);

temp->type = type;

temp->line = line;

temp->next = NULL;

while(p->next!=NULL)

{

p = p->next;

}

p->next = temp;

}

int createNewIden(char \* content,char \*descirbe,int type,int addr,int line)

{

IdentiferNode \* p = idenHead;

IdentiferNode \* temp = new IdentiferNode();

int flag = 0;

int addr\_temp = -2;

while(p->next!=NULL)

{

if(strcmp(content,p->next->content) == 0)

{

flag = 1;

addr\_temp = p->next->addr;

}

p = p->next;

}

if(flag == 0)

{

addr\_temp = ++static\_iden\_number;

}

strcpy(temp->content,content);

strcpy(temp->describe,descirbe);

temp->type = type;

temp->addr = addr\_temp;

temp->line = line;

temp->next = NULL;

p->next = temp;

temp->prev = p;

idenTail = temp;

return addr\_temp;

}

NormalNode\* printNodeLink()

{

NormalNode \* p = normalHead;

NormalNode \* SyntaxPartHead = normalHead;

p = p->next;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Lexical Analysis Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl<<endl;

cout<<setw(30)<<"Lexeme"<<setw(10)<<"\t\tToken"<<"\t\t\t"<<"Line"<<endl;

while(p!=NULL)

{

if(p->type == IDENTIFER)

{

cout<<setw(30)<<p->content<<"\t\t"<<setw(10)<<p->describe<<"\t\t\t"<<p->line<<endl;

}

else

{

cout<<setw(30)<<p->content << "\t\t"<<setw(10) << p->describe<<"\t\t\t"<<p->line<<endl;

}

p = p->next;

}

cout<<endl<<endl;

return SyntaxPartHead;

}

void printErrorLink(bool &TokenError)

{

ErrorNode \* p = errorHead;

if (p->next != NULL) {

TokenError = true;

}

else TokenError = false;

p = p->next;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Error Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl<<endl;

cout<<setw(10)<<"Lexeme"<<setw(30)<<"\t\t\t\t\tToken"<<"\t\t\t"<<"Line"<<endl;

while(p!=NULL)

{

cout<<setw(10)<<p->content<< "\t\t\t\t\t\t" <<setw(30)<< p->describe<<"\t\t\t"<<p->line<<endl;

p = p->next;

}

cout<<endl<<endl;

}

void printIdentLink()

{

IdentiferNode \* p = idenHead;

p = p->next;

cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Identifiers Table\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl<<endl;

cout<<setw(30)<<"Lexeme"<<setw(10)<<"\tToken"<<"\t\t"<<"Address"<<" \t\t"<<"Line"<<endl;

while(p!=NULL)

{

cout<<setw(30)<<p->content << "\t\t" <<setw(10)<<p->describe<<"\t\t"<<p->addr<<"\t\t\t"<<p->line<<endl;

p = p->next;

}

cout<<endl<<endl;

}

int mystrlen(char \* word)

{

if(\*word == '\0')

{

return 0;

}

else

{

return 1+mystrlen(word+1);

}

}

void preProcess(char \* word,int line)

{

const char \* include\_temp = "include";

const char \* define\_temp = "define";

char \* p\_include,\*p\_define;

int flag = 0;

p\_include = strstr(word,include\_temp);

if(p\_include!=NULL)

{

flag = 1;

int i;

for(i=7;;)

{

if(\*(p\_include+i) == ' ' || \*(p\_include+i) == '\t')

{

i++;

}

else

{

break;

}

}

createNewNode(p\_include+i,HEADER\_DESC,HEADER,-1,line);

}

else

{

p\_define = strstr(word,define\_temp);

if(p\_define!=NULL)

{

flag = 1;

int i;

for(i=7;;)

{

if(\*(p\_define+i) == ' ' || \*(p\_define+i) == '\t')

{

i++;

}

else

{

break;

}

}

createNewNode(p\_define+i,CONSTANT\_DESC,MACRO\_VAL,-1,line);

}

}

if(flag == 0)

{

}

}

void close()

{

//delete idenHead;

//delete errorHead;

//delete normalHead;

}

int seekKey(char \* word)

{

for(int i=0; i<keyMap.size(); i++)

{

if(strcmp(word,keyMap[i].first) == 0)

{

return i+1;

}

}

return IDENTIFER;

}

void scanner()

{

char filename[30];

char ch;

char array[30];

char \* word;

int i;

int line = 1;

FILE \* infile;

printf("Please Type in the File name:\n");

scanf("%s",filename);

infile = fopen(filename,"r");

while(!infile)

{

printf("Fail to open file！\n");

return;

}

ch = fgetc(infile);

BIGLOOP: while(ch!=EOF)

{

i = 0;

//Identifiers should begin with letter, can end with '$'

if((ch>='A' && ch<='Z') || (ch>='a' && ch<='z')) {

array[i++] = ch;

ch = fgetc(infile);

while((ch>='A' && ch<='Z')||(ch>='a' && ch<='z')||(ch>='0' && ch<='9') || ch == '$') {

//if the second place of a variable is '$'

if (ch == '$') {

array[i++] = ch;

ch = fgetc(infile);

//if there's more letters or numbers or '$' after '$', which the variable does not end with '$' but contains '$'

if ((ch>='A' && ch<='Z')||(ch>='a' && ch<='z')||(ch>='0' && ch<='9') || ch == '$' || ch == '@'

|| ch == '#' || ch == '?' || ch == '\_') {

while ((ch>='A' && ch<='Z')||(ch>='a' && ch<='z')||(ch>='0' && ch<='9') || ch == '$' || ch == '@'

|| ch == '#' || ch == '?' || ch == '\_') {

array[i++] = ch;

ch = fgetc(infile);

}

fseek(infile,-1L,SEEK\_CUR);//go back one place

word = new char[i+1];

memcpy(word,array,i);

word[i] = '\0';

createNewError(word,IDENTIFIER\_ERROR,IDENTIFIER\_ERROR\_NUM,line);

ch = fgetc(infile);

goto BIGLOOP;

}

//Variable ends with '$', then create a new lexeme

else {

word = new char[i+1];

memcpy(word,array,i);

word[i] = '\0';

int seekTemp = seekKey(word);

if(seekTemp!=IDENTIFER) {

createNewNode(word,KEY\_DESC,seekTemp,-1,line);

}

else {

int addr\_tmp = createNewIden(word,IDENTIFER\_DESC,seekTemp,-1,line);

createNewNode(word,IDENTIFER\_DESC,seekTemp,addr\_tmp,line);

}

fseek(infile,-1L,SEEK\_CUR);//go back one place

ch = fgetc(infile);

goto BIGLOOP;

}

}

array[i++] = ch;

ch = fgetc(infile);

}

word = new char[i+1];

memcpy(word,array,i);

word[i] = '\0';

int seekTemp = seekKey(word);

if (array[i - 1] >='0' && array[i - 1]<='9') {

createNewError(word,IDENTIFIER\_ERROR,IDENTIFIER\_ERROR\_NUM,line);

ch = fgetc(infile);

goto BIGLOOP;

}

else if(seekTemp!=IDENTIFER) {

createNewNode(word,KEY\_DESC,seekTemp,-1,line);

}

else {

int addr\_tmp = createNewIden(word,IDENTIFER\_DESC,seekTemp,-1,line);

createNewNode(word,IDENTIFER\_DESC,seekTemp,addr\_tmp,line);

}

fseek(infile,-1L,SEEK\_CUR);//go back one place

}

//if a variable starts with '$', the entire variable is

else if (ch == '$' || ch == '\_') {

array[i++] = ch;

ch = fgetc(infile);

if ((ch>='A' && ch<='Z')||(ch>='a' && ch<='z')||(ch>='0' && ch<='9') || ch == '$' || ch == '@'

|| ch == '#' || ch == '?' || ch == '\_') {

while ((ch>='A' && ch<='Z')||(ch>='a' && ch<='z')||(ch>='0' && ch<='9') || ch == '$' || ch == '@'

|| ch == '#' || ch == '?' || ch == '\_') {

array[i++] = ch;

ch = fgetc(infile);

}

fseek(infile,-1L,SEEK\_CUR);//go back one place

word = new char[i+1];

memcpy(word,array,i);

word[i] = '\0';

createNewError(word,IDENTIFIER\_ERROR,IDENTIFIER\_ERROR\_NUM,line);

ch = fgetc(infile);

goto BIGLOOP;

}

}

//Start with number

else if(ch >='0' && ch<='9')

{

int flag = 0;

int flag2 = 0;

//deal with int

while(ch >='0' && ch<='9')

{

array[i++] = ch;

ch = fgetc(infile);

}

//deal with float which is the same as real

if(ch == '.')

{

flag2 = 1;

array[i++] = ch;

ch = fgetc(infile);

if(ch>='0' && ch<='9')

{

while(ch>='0' && ch<='9')

{

array[i++] = ch;

ch = fgetc(infile);

}

}

else

{

flag = 1;

}

}

word = new char[i+1];

memcpy(word,array,i);

word[i] = '\0';

if(flag == 1)

{

createNewError(word,REAL\_ERROR,REAL\_ERROR\_NUM,line);

}

else if(flag == 2)

{

createNewError(word,REAL\_ERROR,REAL\_ERROR\_NUM,line);

}

else

{

if(flag2 == 0)

{

createNewNode(word,INT\_DESC,INT\_VAL,-1,line);

}

else

{

createNewNode(word,REAL\_DESC,REAL\_VAL,-1,line);

}

}

fseek(infile,-1L,SEEK\_CUR);

}

// //start with "/"

// else if(ch == '/')

// {

// ch = fgetc(infile);

// createNewNode("/",OPE\_DESC,DIV,-1,line);

// }

else if(ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n')

{

if(ch == '\n')

{

line++;

}

}

else

{

if(ch == EOF)

{

return;

}

else if(ch == '-')

{

array[i++] = ch;

ch = fgetc(infile);

createNewNode("-",OPE\_DESC,SUB,-1,line);

fseek(infile,-1L,SEEK\_CUR);

}

else if (ch == '/') {

ch = fgetc(infile);

createNewNode("/", OPE\_DESC, DIV, -1, line);

fseek(infile, -1L, SEEK\_CUR);

}

else if(ch == '+')

{

ch = fgetc(infile);

createNewNode("+",OPE\_DESC,ADD,-1,line);

fseek(infile,-1L,SEEK\_CUR);

}

else if(ch == '\*')

{

ch = fgetc(infile);

createNewNode("\*",OPE\_DESC,MUL,-1,line);

fseek(infile,-1L,SEEK\_CUR);

}

else if(ch == '^')

{

ch = fgetc(infile);

if(ch == '=')

{

createNewNode("^=",OPE\_DESC,COMPLETE\_BYTE\_XOR,-1,line);

}

}

//deal with %%

else if(ch == '%')

{

ch = fgetc(infile);

if(ch == '%')

{

createNewNode("%%",CLE\_OPE\_DESC,PERCENTPERCENT,-1,line);

}

}

//deal with comments

else if (ch == '!') {

ch = fgetc(infile);

while (1) {

if (ch == '\n') {

line++;

}

ch = fgetc(infile);

if (ch == EOF) {

createNewError(\_NULL, NOTE\_ERROR, NOTE\_ERROR\_NUM, line);

return ;

}

if (ch == '!') {

break;

}

}

}

//Deal with "<"

else if(ch == '<')

{

ch = fgetc(infile);

createNewNode("<",OPE\_DESC,LES\_THAN,-1,line);

fseek(infile,-1L,SEEK\_CUR);

}

//Deal with ">"

else if(ch == '>')

{

ch = fgetc(infile);

createNewNode(">",OPE\_DESC,GRT\_THAN,-1,line);

fseek(infile,-1L,SEEK\_CUR);

}

else if(ch == '=')

{

ch = fgetc(infile);

if(ch == '=')

{

createNewNode("==",OPE\_DESC,EQUAL,-1,line);

}

else if (ch == '>') {

createNewNode("=>",OPE\_DESC,GRT\_EQUAL,-1,line);

}

else if (ch == '<') {

createNewNode("=<",OPE\_DESC,LES\_EQUAL,-1,line);

}

else

{

createNewNode("=",OPE\_DESC,ASG,-1,line);

fseek(infile,-1L,SEEK\_CUR);

}

}

else if(ch == '(')

{

leftSmall++;

lineBra[0][leftSmall] = line;

createNewNode("(",CLE\_OPE\_DESC,LEFT\_BRA,-1,line);

}

else if(ch == ')')

{

rightSmall++;

lineBra[1][rightSmall] = line;

createNewNode(")",CLE\_OPE\_DESC,RIGHT\_BRA,-1,line);

}

else if(ch == '[')

{

leftMiddle++;

lineBra[2][leftMiddle] = line;

createNewNode("[",CLE\_OPE\_DESC,LEFT\_INDEX,-1,line);

}

else if(ch == ']')

{

rightMiddle++;

lineBra[3][rightMiddle] = line;

createNewNode("]",CLE\_OPE\_DESC,RIGHT\_INDEX,-1,line);

}

else if(ch == '{')

{

leftBig++;

lineBra[4][leftBig] = line;

createNewNode("{",CLE\_OPE\_DESC,L\_BOUNDER,-1,line);

}

else if(ch == '}')

{

rightBig++;

lineBra[5][rightBig] = line;

createNewNode("}",CLE\_OPE\_DESC,R\_BOUNDER,-1,line);

}

else if(ch == '.')

{

createNewNode(".",CLE\_OPE\_DESC,POINTER,-1,line);

}

else if(ch == ',')

{

createNewNode(",",CLE\_OPE\_DESC,COMMA,-1,line);

}

else if (ch == ':') {

createNewNode(":",CLE\_OPE\_DESC,COLON,-1,line);

}

else if(ch == ';')

{

createNewNode(";",CLE\_OPE\_DESC,SEMI,-1,line);

}

else

{

char temp[2];

temp[0] = ch;

temp[1] = '\0';

createNewError(temp,CHAR\_ERROR,CHAR\_ERROR\_NUM,line);

}

}

ch = fgetc(infile);

}

}

void BraMappingError()

{

if(leftSmall != rightSmall)

{

int i = (leftSmall>rightSmall) ? (leftSmall-rightSmall) : (rightSmall - leftSmall);

bool flag = (leftSmall>rightSmall) ? true : false;

if(flag)

{

while(i--)

{

createNewError(\_NULL,LEFT\_BRA\_ERROR,LEFT\_BRA\_ERROR\_NUM,lineBra[0][i+1]);

}

}

else

{

while(i--)

{

createNewError(\_NULL,RIGHT\_BRA\_ERROR,RIGHT\_BRA\_ERROR\_NUM,lineBra[1][i+1]);

}

}

}

if(leftMiddle != rightMiddle)

{

int i = (leftMiddle>rightMiddle) ? (leftMiddle-rightMiddle) : (rightMiddle - leftMiddle);

bool flag = (leftMiddle>rightMiddle) ? true : false;

if(flag)

{

while(i--)

{

createNewError(\_NULL,LEFT\_INDEX\_ERROR,LEFT\_INDEX\_ERROR\_NUM,lineBra[2][i+1]);

}

}

else

{

while(i--)

{

createNewError(\_NULL,RIGHT\_INDEX\_ERROR,RIGHT\_INDEX\_ERROR\_NUM,lineBra[3][i+1]);

}

}

}

if(leftBig != rightBig)

{

int i = (leftBig>rightBig) ? (leftBig-rightBig) : (rightBig - leftSmall);

bool flag = (leftBig>rightBig) ? true : false;

if(flag)

{

while(i--)

{

createNewError(\_NULL,L\_BOUNDER\_ERROR,L\_BOUNDER\_ERROR\_NUM,lineBra[4][i+1]);

}

}

else

{

while(i--)

{

createNewError(\_NULL,R\_BOUNDER\_ERROR,R\_BOUNDER\_ERROR\_NUM,lineBra[5][i+1]);

}

}

}

}

//

// SynAnalysis.h

// CPSC323\_HW3

//

// Created by Liren on 4/21/18.

// Copyright © 2018 Liren. All rights reserved.

//

#ifndef SynAnalysis\_h

#define SynAnalysis\_h

bool Rat18S(NormalNode&);

bool OptFunctionDefinitions(NormalNode&);

bool FunctionDefinitions(NormalNode&);

bool Function(NormalNode&);

bool OptParameterList(NormalNode&);

bool ParameterList(NormalNode&);

bool Parameter(NormalNode&);

bool Qualifier(NormalNode&);

bool Body(NormalNode&);

bool OptDeclarationList(NormalNode&);

bool DeclarationList(NormalNode&);

bool Declaration(NormalNode&);

bool IDs(NormalNode&);

bool StatementList(NormalNode&);

bool Statement(NormalNode&);

bool Compound(NormalNode&);

bool Assign(NormalNode&);

bool If(NormalNode&);

bool Return(NormalNode&);

bool Print(NormalNode&);

bool Scan(NormalNode&);

bool While(NormalNode&);

bool Condition(NormalNode&);

bool Relop(NormalNode&);

bool Expression(NormalNode&);

bool Term(NormalNode&);

bool Factor(NormalNode&);

bool Primary(NormalNode&);

bool Identifier(NormalNode&);

bool Real(NormalNode&);

bool Integer(NormalNode&);

bool Empty(NormalNode&);

bool ExpressionP(NormalNode&);

bool TermP(NormalNode&);

bool FunctionDefinitionsP(NormalNode&);

bool ParameterListP(NormalNode&);

bool DeclarationListP(NormalNode&);

bool IDsP(NormalNode&);

bool StatementListP(NormalNode&);

bool IfP(NormalNode&);

bool ReturnP(NormalNode&);

bool PrimaryP(NormalNode&);

void analysis(NormalNode&);

#endif /\* SynAnalysis\_h \*/

//

// SynAnalysis.cpp

// CPSC323\_HW3

//

// Created by Liren on 4/21/18.

// Copyright © 2018 Liren. All rights reserved.

//

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fstream>

#include <vector>

#include <string>

#include <stack>

#include <iomanip>

#include "LexAnalysis.h"

#include "SynAnalysis.h"

using namespace std;

extern NormalNode \* normalHead;//First node

extern NormalNode \* normalTail;//Last node

extern vector<pair<const char \*,int> > keyMap;

extern vector<pair<const char \*,int> > operMap;

extern vector<pair<const char \*,int> > limitMap;

vector<pair<const char \*,int> > specialMap;//special symbol in grammar (-> | EPSILON $)

fstream resultfile;

string SYMBOL\_TYPE = "";

stack<int> jumpStack;

struct Instruction\_Table {

int inst\_address\_;

string op\_;

string oprnd\_;

};

struct Symbol\_Table {

string symbol\_;

string type\_;

int address\_;

};

int instr\_address = 1;

int symbol\_table\_num = 1;

int curr\_Symbol\_addr = 2000;

Instruction\_Table\* Inst\_table = new Instruction\_Table[500]();

int Inst\_table\_size = 0;

Symbol\_Table\* Sym\_table = new Symbol\_Table[100]();

int Sym\_table\_size = 1;

void gen\_instr(string op, string oprnd) {

Inst\_table[instr\_address].inst\_address\_ = instr\_address;

Inst\_table[instr\_address].op\_ = op;

Inst\_table[instr\_address].oprnd\_ = oprnd;

instr\_address++;

}

int get\_address(NormalNode node) {

return Sym\_table[node.addr].address\_;

}

void back\_patch(int instr\_address) {

int addr = 0;

addr = jumpStack.top();

jumpStack.pop();

Inst\_table[addr].oprnd\_ = to\_string(instr\_address);

}

bool Rat18S(NormalNode& p) {

NormalNode iterator = p;

if (!OptFunctionDefinitions(iterator)) return false;

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str(iterator.content);

if (str.compare("%%") != 0) {

cout << "Syntax Error: " << "Line " << p.line << " , %% not matched." << endl;

return false;

}

if (iterator.next == NULL) {

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!OptDeclarationList(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!StatementList(iterator)) {

return false;

}

p = iterator;

cout << "Line " << p.line << " , <Rat18S> matched." << endl;

return true;

}

bool OptFunctionDefinitions(NormalNode& p) {

NormalNode iterator = p;

if (FunctionDefinitions(iterator)) {

cout << "Line " << p.line << " , <OptFunctionDefinitions> -> <OptFunctionDefinitions> is match." << endl;

p = iterator;

return true;

}

else {

cout << "Line " << p.line << " , <OptFunctionDefinitions> -> <Empty>" << endl;

p = \*p.prev;

return true;

}

}

bool FunctionDefinitions(NormalNode& p) {

NormalNode iterator = p;

if (!Function(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!FunctionDefinitionsP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <FunctionDefinitions> -> <Function><FunctionDefinitionsP> is match." << endl;

p = iterator;

return true;

}

bool Function(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("function") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Identifier(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("[") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!OptParameterList(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str3(p.content);

if (str3.compare("]") != 0) {

cout << "Syntax Error: " << "Line " << p.line << " , seperator \"]\" not matched." << endl;

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!OptDeclarationList(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Body(iterator)) {

return false;

}

cout << "Line " << p.line << " , <Function> -> function <Identifier> [ <OptParameterList> ] <OptDeclarationList> <Body> matched "<< endl;

p = iterator;

return true;

}

bool OptParameterList(NormalNode& p) {

NormalNode iterator = p;

if (ParameterList(iterator)) {

cout << "Line " << p.line << " , <OptParameterList> -> <ParameterList> matched." << endl;

p = iterator;

return true;

}

else {

cout << "Line " << p.line << " , <OptParameterList> -> <Empty>" << endl;

p = \*p.prev;

return true;

}

}

bool ParameterList(NormalNode& p) {

NormalNode iterator = p;

if (!Parameter(iterator)) {

cout << "Syntax Error: " << "Line " << p.line << " , <Parameter> not matched." << endl;

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!ParameterListP(iterator)) {

cout << "Syntax Error: " << "Line " << p.line << " , <ParameterListP> not matched." << endl;

return false;

}

cout << "Line " << p.line << " , <ParameterList> -> <Parameter> <ParameterListP> matched." << endl;

p = iterator;

return true;

}

bool Parameter(NormalNode& p) {

NormalNode iterator = p;

if (!IDs(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str(p.content);

if (str.compare(":") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Qualifier(iterator)) {

return false;

}

cout << "Line " << p.line << " , <Parameter> -> <IDs> : <Qualifier> matched." << endl;

p = iterator;

return true;

}

bool Qualifier(NormalNode& p) {

string str(p.content);

if (str.compare("int") == 0) {

cout << "Line " << p.line << " , <Qualifier> -> int matched." << endl;

return true;

}

else if (str.compare("boolean") == 0) {

cout << "Line " << p.line << " , <Qualifier> -> boolean matched." << endl;

return true;

}

else if (str.compare("real") == 0) {

cout << "Line " << p.line << " , <Qualifier> -> real matched." << endl;

return true;

}

else {

return false;

}

}

bool Body(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("{") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!StatementList(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("}") != 0) {

cout << "Syntax Error: " << "Line " << p.line << " , seperator \"}\" not matched." << endl;

return false;

}

cout << "Line " << p.line << " , <Body> -> { <StatementList> } matched." << endl;

p = iterator;

return true;

}

bool OptDeclarationList(NormalNode& p) {

NormalNode iterator = p;

if (DeclarationList(iterator)) {

cout << "Line " << p.line << " , <OptDeclarationList> -> <DeclarationList> matched." << endl;

p = iterator;

return true;

}

else {

cout << "Line " << p.line << " , <OptDeclarationList> -> <Empty>." << endl;

p = \*p.prev;

return true;

}

}

bool DeclarationList(NormalNode& p) {

NormalNode iterator = p;

if (!Declaration(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str(p.content);

if (str.compare(";") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!DeclarationListP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <DeclarationList> -> <Declaration> ; <DeclarationListP> matched." << endl;

p = iterator;

return true;

}

bool Declaration(NormalNode& p) {

NormalNode iterator = p;

string str(iterator.content);

if (str.compare("int") == 0) {

SYMBOL\_TYPE = "int";

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

NormalNode declarationStart = p;

if (!IDs(iterator)) {

return false;

}

else {

NormalNode declarationEnd = iterator;

while (declarationStart.next != declarationEnd.next) {

string declarationStartString(declarationStart.describe);

if (declarationStartString.compare("Identifier") == 0) {

//check duplicate declaration

if (Sym\_table[declarationStart.addr].address\_ != 0) {

cout << "Syntax Error: \"" << declarationStart.content << "\" is already defined." << endl;

return false;

}

//create declared variables in symbol table

string declarationStartContent(declarationStart.content);

Sym\_table[declarationStart.addr].type\_ = SYMBOL\_TYPE;

Sym\_table[declarationStart.addr].address\_ = declarationStart.addr - 1 + 2000;

Sym\_table[declarationStart.addr].symbol\_ = declarationStartContent;

}

declarationStart = \*declarationStart.next;

}

string declarationStartString(declarationStart.describe);

if (declarationStartString.compare("Identifier") == 0) {

//check duplicate declaration

if (Sym\_table[declarationStart.addr].address\_ != 0) {

cout << "Syntax Error: \"" << declarationStart.content << "\" is already defined." << endl;

return false;

}

string declarationStartContent(declarationStart.content);

Sym\_table[declarationStart.addr].type\_ = SYMBOL\_TYPE;

Sym\_table[declarationStart.addr].address\_ = declarationStart.addr - 1 + 2000;

Sym\_table[declarationStart.addr].symbol\_ = declarationStartContent;

}

cout << "Line " << p.line << " , <Declaration> -> int <IDs> matched." << endl;

p = iterator;

return true;

}

}

else if (str.compare("boolean") == 0) {

SYMBOL\_TYPE = "boolean";

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!IDs(iterator)) {

return false;

}

else {

cout << "Line " << p.line << " , <Declaration> -> boolean <IDs> matched." << endl;

p = iterator;

return true;

}

}

else if (str.compare("real") == 0) {

cout << "No \"real\" type variable is allowed in simplified Rat18S compiler." << endl;

return false;

SYMBOL\_TYPE = "real";

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!IDs(iterator)) {

return false;

}

else {

cout << "Line " << p.line << " , <Declaration> -> real <IDs> matched." << endl;

p = iterator;

return true;

}

}

else {

//cout << "Syntax Error: " << "Line " << p.line << " , <Declaration> not matched." << endl;

return false;

}

}

bool IDs(NormalNode& p) {

NormalNode iterator = p;

if (!Identifier(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!IDsP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <IDs> -> <Identifier> <IDsP> matched." << endl;

p = iterator;

return true;

}

bool StatementList(NormalNode& p) {

NormalNode iterator = p;

if (!Statement(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!StatementListP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <StatementList> -> <Statement> <StatementListP> matched." << endl;

p = iterator;

return true;

}

bool Statement(NormalNode& p) {

NormalNode iterator = p;

if (Compound(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <Compound> matched." << endl;

p = iterator;

return true;

}

else if (Assign(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <Assign> matched." << endl;

p = iterator;

return true;

}

else if (If(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <If> matched." << endl;

p = iterator;

return true;

}

else if (Return(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <Return> matched." << endl;

p = iterator;

return true;

}

else if (Print(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <Print> matched." << endl;

p = iterator;

return true;

}

else if (Scan(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <Scan> matched." << endl;

p = iterator;

return true;

}

else if (While(iterator)) {

cout << "Line " << p.line << " , <Statement> -> <While> matched." << endl;

p = iterator;

return true;

}

else {

//cout << "Syntax Error: " << "Line " << p.line << " , <Statement> not matched." << endl;

return false;

}

}

bool Compound(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("{") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!StatementList(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("}") != 0) {

return false;

}

cout << "Line " << p.line << " , <Compound> -> { <StatementList> } matched." << endl;

p = iterator;

return true;

}

bool Assign(NormalNode& p) {

NormalNode iterator = p;

if (!Identifier(iterator)) {

return false;

}

NormalNode save = iterator;

string pContent(save.content);

string pDescribe(save.describe);

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str(p.content);

if (str.compare("=") != 0) {

cout << "Line: " << p.line << "Syntax Error: " << "= expected" << endl;

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Expression(iterator)) {

return false;

}

string saveContent(save.content);

Sym\_table[save.addr].type\_ = SYMBOL\_TYPE;

Sym\_table[save.addr].address\_ = save.addr - 1 + 2000;

Sym\_table[save.addr].symbol\_ = saveContent;

gen\_instr("POPM", to\_string(get\_address(save)));

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare(";") != 0) {

return false;

}

cout << "Line " << p.line << " , <Assign> -> <Identifier> = <Expression> ; matched." << endl;

p = iterator;

return true;

}

bool If(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("if") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("(") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Condition(iterator)) {

return false;

}

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str3(p.content);

if (str3.compare(")") != 0) {

cout << "Line: " << p.line << " Syntax Error: " << ") expected." << endl;

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Statement(iterator)) {

return false;

}

back\_patch(instr\_address);

p = iterator;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!IfP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <If> -> if ( <Condition> ) <Statement> <IfP> matched." << endl;

p = iterator;

return true;

}

bool Return(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("return") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!ReturnP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <Return> -> return <ReturnP> matched." << endl;

p = iterator;

return true;

}

bool Print(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("put") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("(") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

//

// NormalNode printIDstart = iterator;

if (!Expression(iterator)) {

return false;

}

gen\_instr("STDOUT", " ");

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str3(p.content);

if (str3.compare(")") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str4(p.content);

if (str4.compare(";") != 0) {

return false;

}

cout << "Line " << p.line << " , <Print> -> put ( <Expression> ) ; matched." << endl;

p = iterator;

return true;

}

bool Scan(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("get") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("(") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

NormalNode scanIDstart = p;

if (!IDs(iterator)) {

return false;

}

NormalNode scanIDend = iterator;

while (scanIDstart.next != scanIDend.next) {

string scanIDstartString(scanIDstart.describe);

if (scanIDstartString.compare("Identifier") == 0) {

gen\_instr("STDIN", " ");

gen\_instr("POPM", to\_string(get\_address(scanIDstart)));

}

else if (scanIDstartString.compare("int") == 0) {

gen\_instr("STDIN", " ");

gen\_instr("POPM", to\_string(get\_address(scanIDstart)));

}

scanIDstart = \*scanIDstart.next;

}

string scanIDstartString(scanIDstart.describe);

if (scanIDstartString.compare("Identifier") == 0) {

gen\_instr("STDIN", " ");

gen\_instr("POPM", to\_string(get\_address(scanIDstart)));

}

else if (scanIDstartString.compare("int") == 0) {

gen\_instr("STDIN", " ");

gen\_instr("POPM", to\_string(get\_address(scanIDstart)));

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str3(p.content);

if (str3.compare(")") != 0) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str4(p.content);

if (str4.compare(";") != 0) {

return false;

}

cout << "Line " << p.line << " , <Scan> -> get ( <IDs> ) ; matched." << endl;

p = iterator;

return true;

}

bool While(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("while") != 0) {

cout << "Line: " << p.line << " Syntax Error: " << "while expected" << endl;

return false;

}

int addr = instr\_address;

gen\_instr("LABEL", " ");

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("(") != 0) {

cout << "Line: " << p.line << " Syntax Error: " << "( expected" << endl;

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Condition(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str3(p.content);

if (str3.compare(")") != 0) {

cout << "Line: " << p.line << " Syntax Error: " << ") expected" << endl;

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Statement(iterator)) {

return false;

}

gen\_instr("JUMP", to\_string(addr));

back\_patch(instr\_address);

cout << "Line " << p.line << " , <While> -> while ( <Condition> ) <Statement> matched." << endl;

p = iterator;

return true;

}

bool Condition(NormalNode& p) {

NormalNode iterator = p;

if (!Expression(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Relop(iterator)) {

return false;

}

NormalNode op = iterator;

string opstring(op.content);

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Expression(iterator)) {

return false;

}

if (opstring.compare("<") == 0) {

gen\_instr("LES", " ");

jumpStack.push(instr\_address);

gen\_instr("JUMPZ", " ");

}

else if (opstring.compare(">") == 0) {

gen\_instr("GRT", " ");

jumpStack.push(instr\_address);

gen\_instr("JUMPZ", " ");

}

else if (opstring.compare("==") == 0) {

gen\_instr("EQU", " ");

jumpStack.push(instr\_address);

gen\_instr("JMPZ", " ");

}

else if (opstring.compare("^=") == 0) {

gen\_instr("NEQ", " ");

jumpStack.push(instr\_address);

gen\_instr("JUMPZ", " ");

}

else if (opstring.compare("=>") == 0) {

gen\_instr("GEQ", " ");

jumpStack.push(instr\_address);

gen\_instr("JUMPZ", " ");

}

else if (opstring.compare("=<") == 0) {

gen\_instr("LEQ", " ");

jumpStack.push(instr\_address);

gen\_instr("JUMPZ", " ");

}

cout << "Line " << p.line << " , <Condition> -> <Expression> <Relop> <Expression> matched." << endl;

p = iterator;

return true;

}

bool Relop(NormalNode& p) {

string str(p.content);

if (str.compare("==") == 0) {

cout << "Line " << p.line << " , <Relop> -> == matched." << endl;

return true;

}

else if (str.compare("^=") == 0) {

cout << "Line " << p.line << " , <Relop> -> ^= matched." << endl;

return true;

}

else if (str.compare("<") == 0) {

cout << "Line " << p.line << " , <Relop> -> < matched." << endl;

return true;

}

else if (str.compare(">") == 0) {

cout << "Line " << p.line << " , <Relop> -> > matched." << endl;

return true;

}

else if (str.compare("=>") == 0) {

cout << "Line " << p.line << " , <Relop> -> => matched." << endl;

return true;

}

else if (str.compare("=<") == 0) {

cout << "Line " << p.line << " , <Relop> -> =< matched." << endl;

return true;

}

else {

cout << "Line: " << p.line << " Syntax Error: " << "==, < , >, =>, =< or ^= token is expcted. " << endl;

return false;

}

}

bool Expression(NormalNode& p) {

NormalNode iterator = p;

if (!Term(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!ExpressionP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <Expression> -> <Term> <ExpressionP> matched." << endl;

p = iterator;

return true;

}

bool Term(NormalNode& p) {

NormalNode iterator = p;

if (!Factor(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!TermP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <Term> -> <Factor> <TermP> matched." << endl;

p = iterator;

return true;

}

bool Factor(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("-") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Primary(iterator)) {

return false;

}

else {

cout << "Line " << p.line << " , <Factor> -> - <Primary> matched." << endl;

p = iterator;

return true;

}

}

if (Primary(iterator)) {

cout << "Line " << p.line << " , <Factor> -> <Primary> matched." << endl;

p = iterator;

return true;

}

cout << "Line: " << p.line << " Syntax Error: id expected" << endl;

return false;

}

bool Primary(NormalNode& p) {

NormalNode iterator = p;

string str(iterator.content);

if (Identifier(iterator)) {

if (Sym\_table[iterator.addr].address\_ == 0) {

cout << "Error: " << iterator.content << " is not declared." << endl;

return false;

}

else {

gen\_instr("PUSHM", to\_string(get\_address(p)));

}

//gen\_instr("PUSHM", to\_string(get\_address(p)));

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (PrimaryP(iterator)) {

cout << "Line " << p.line << " , <Primary> -> <Identifier> <PrimaryP> matched." << endl;

p = iterator;

return true;

}

else {

return false;

}

}

else if (Integer(iterator)) {

cout << "Line " << p.line << " , <Primary> -> <Integer> matched." << endl;

p = iterator;

return true;

}

else if (str.compare("(") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Expression(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(iterator.content);

if (str2.compare(")") != 0) {

return false;

}

cout << "Line " << p.line << " , <Primary> -> ( <Expression> ) matched." << endl;

p = iterator;

return true;

}

else if (Real(iterator)) {

cout << "Line " << p.line << " , <Primary> -> <Real> matched." << endl;

p = iterator;

return true;

}

else if (str.compare("true") == 0) {

cout << "Line " << p.line << " , <Primary> -> true matched." << endl;

p = iterator;

return true;

}

else if (str.compare("false") == 0) {

cout << "Line " << p.line << " , <Primary> -> false matched." << endl;

p = iterator;

return true;

}

else {

return false;

}

}

bool Identifier(NormalNode& p) {

string str(p.describe);

if (str.compare("Identifier") == 0 ) {

cout << "Line " << p.line << ", id = " << p.content << " , <Identifier> -> id matched." << endl;

return true;

}

return false;

}

bool Real(NormalNode& p) {

string str(p.describe);

if (str.compare("real") == 0) {

//cout << "Line " << p.line << " , <Real> -> real matched." << endl;

cout << "No real type is allowed in Rat18S compiler." << endl;

return false;

}

return false;

}

bool Integer(NormalNode& p) {

string str(p.describe);

if (str.compare("int") == 0) {

string pContent(p.content);

Sym\_table[p.addr].type\_ = SYMBOL\_TYPE;

Sym\_table[p.addr].address\_ = atoi(p.content);

Sym\_table[p.addr].symbol\_ = pContent;

gen\_instr("PUSHI", to\_string(get\_address(p)));

cout << "Line " << p.line << " , <Integer> -> int matched." << endl;

return true;

}

return false;

}

bool Empty(NormalNode& p) {

cout << "Line " << p.line << " , <Empty> -> EPSILON matched." << endl;

return true;

}

bool ExpressionP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("+") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Term(iterator)) {

return false;

}

gen\_instr("ADD", " ");

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!ExpressionP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <ExpressionP> -> + <Term> <ExpressionP> matched." << endl;

p = iterator;

return true;

}

else if (str.compare("-") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Term(iterator)) {

return false;

}

gen\_instr("SUB", " ");

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!ExpressionP(iterator)) {

return false;

}

cout << "Line " << p.line << " , <ExpressionP> -> - <Term> <ExpressionP> matched." << endl;

p = iterator;

return true;

}

else {

cout << "Line " << p.line << " , <ExpressionP> -> EPSILON matched." << endl;

p = \*p.prev;

return true;

}

}

bool TermP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("\*") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Factor(iterator)) {

return false;

}

gen\_instr("MUL", " ");

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!TermP(iterator)) {

return false;

}

else {

p = iterator;

cout << "Line " << p.line << " , <TermP> -> \* <Factor> <TermP> matched." << endl;

return true;

}

}

else if (str.compare("/") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Factor(iterator)) {

return false;

}

gen\_instr("DIV", " ");

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!TermP(iterator)) {

return false;

}

else {

p = iterator;

cout << "Line " << p.line << " , <TermP> -> / <Factor> <TermP> matched." << endl;

return true;

}

}

else {

cout << "Line " << p.line << " , <TermP> -> EPSILON matched." << endl;

p = \*p.prev;

return true;

}

}

bool FunctionDefinitionsP(NormalNode& p) {

NormalNode iterator = p;

if (FunctionDefinitions(iterator)) {

p = iterator;

cout << "Line " << p.line << " , <FunctionDefinitionsP> -> <FunctionDefinitions> matched." << endl;

return true;

}

else {

p = \*p.prev;

cout << "Line " << p.line << " , <FunctionDefinitionsP> -> EPSILON matched." << endl;

return true;

}

}

bool ParameterListP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare(",") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!ParameterList(iterator)) {

return false;

}

else {

p = iterator;

cout << "Line " << p.line << " , <ParameterListP> -> <ParameterList> matched." << endl;

return true;

}

}

else {

p = \*p.prev;

cout << "Line " << p.line << " , <ParameterListP> -> EPSILON matched." << endl;

return true;

}

}

bool DeclarationListP(NormalNode& p) {

NormalNode iterator = p;

if (DeclarationList(iterator)) {

p = iterator;

cout << "Line " << p.line << " , <DeclarationListP> -> <DeclarationList> matched." << endl;

return true;

}

else {

p = \*p.prev;

cout << "Line " << p.line << " , <DeclarationListP> -> EPSILON matched." << endl;

return true;

}

}

bool IDsP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare(",") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!IDs(iterator)) {

return false;

}

else {

p = iterator;

cout << "Line " << p.line << " , <IDsP> -> , <IDs> matched." << endl;

return true;

}

}

else {

p = \*p.prev;

cout << "Line " << p.line << " , <IDsP> -> EPSILON matched." << endl;

return true;

}

}

bool StatementListP(NormalNode& p) {

NormalNode iterator = p;

if (StatementList(iterator)) {

p = iterator;

cout << "Line " << p.line << " , <StatementListP> -> <StatementList> matched." << endl;

return true;

}

else {

p = \*p.prev;

cout << "Line " << p.line << " , <StatementListP> -> EPSILON matched." << endl;

return true;

}

}

bool IfP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("endif") == 0) {

cout << "Line " << p.line << " , <IfP> -> endif matched." << endl;

return true;

}

else if (str.compare("else") == 0) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!Statement(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare("endif") != 0) {

cout << "Syntax Error: " << "Line " << p.line << " , \"endif\" is missing." << endl;

return false;

}

p = iterator;

cout << "Line " << p.line << " , <IfP> -> else <Statement> endif matched." << endl;

return true;

}

else {

cout << "Syntax Error: " << "Line " << p.line << " , \"endif or else\" is missing." << endl;

return false;

}

}

bool ReturnP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare(";") == 0) {

cout << "Line " << p.line << " , <ReturnP> -> ; matched." << endl;

return true;

}

else if (Expression(iterator)) {

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare(";") != 0) {

cout << "Syntax Error: " << "Line " << p.line << " , \";\" is missing." << endl;

return false;

}

else {

cout << "Line " << p.line << " , <ReturnP> -> <Expression> matched." << endl;

p = iterator;

return true;

}

}

cout << "Syntax Error: " << "Line " << p.line << " , \";\" is missing." << endl;

return false;

}

bool PrimaryP(NormalNode& p) {

NormalNode iterator = p;

string str(p.content);

if (str.compare("(") == 0) {

cout << "Line " << p.line << " , seperator \"(\" matched." << endl;

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

if (!IDs(iterator)) {

return false;

}

if (iterator.next == NULL) {

p = iterator;

cout << "EOF: " << "Line " << p.line << "." << endl;

return true;

}

p = \*iterator.next;

iterator = p;

string str2(p.content);

if (str2.compare(")") != 0) {

cout << "Syntax Error: " << "Line " << p.line << " , seperator \")\" not matched." << endl;

return false;

}

else {

cout << "Line " << p.line << " , <PrimaryP> -> ( <IDs> ) matched." << endl;

p = iterator;

return true;

}

}

else {

cout << "Line " << p.line << " , <PrimaryP> -> EPSILON is matched." << endl;

p = \*p.prev;

return true;

}

}

void analysis(NormalNode& head) {

NormalNode p = head;

p = \*p.next;

if (Rat18S(p) && p.next == NULL) {

cout << "success" << endl;

gen\_instr("", "");

//Symbol\_Table\* Sym\_table = new Symbol\_Table[100]();

Symbol\_Table\* Sym\_table\_iterator = Sym\_table;

//Instruction\_Table\* Inst\_table = new Instruction\_Table[500]();

Instruction\_Table\* Inst\_table\_iterator = Inst\_table;

cout << endl << endl;

//Analysis Result output file

resultfile.open("Symbol\_Table.txt",std::ios::out);

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Symbol Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

cout << "\t\t" << "Identifier" << " \t\t\t" << "Type" << "\t\t" << "Memory Addr." << endl;

resultfile << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Symbol Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

resultfile << "\t" << "Identifier" << "\t\t" << "Type" << "\t\t" << "Memory Addr." << endl;

for (int i = 1 ; i < 10 ; i++) {

if (Sym\_table\_iterator[i].address\_ == 0) break;

cout << setw(15) << Sym\_table\_iterator[i].symbol\_ << setw(20) << Sym\_table\_iterator[i].type\_ << setw(15) << Sym\_table\_iterator[i].address\_ << endl;

resultfile << setw(15) << Sym\_table\_iterator[i].symbol\_ << setw(20) << Sym\_table\_iterator[i].type\_ << setw(20) << Sym\_table\_iterator[i].address\_ << endl;

}

cout << endl;

resultfile.close();

resultfile.open("Instruction\_Table.txt",std::ios::out);

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Instruction Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

cout << "\t\t" << "Instruction Addr." << " \t\t\t" << "Operation" << "\t\t" << "Operand" << endl;

resultfile << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Instruction Table \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << endl;

resultfile << "\t\t" << "Instruction Addr." << " \t\t" << "Operation" << "\t\t" << "Operand" << endl;

for (int i = 1 ; i < instr\_address; i++) {

cout << setw(18) << Inst\_table\_iterator[i].inst\_address\_ << setw(26) << Inst\_table\_iterator[i].op\_ << setw(14) << Inst\_table\_iterator[i].oprnd\_ << endl;

resultfile << setw(18) << Inst\_table\_iterator[i].inst\_address\_ << setw(35) << Inst\_table\_iterator[i].op\_ << setw(24) << Inst\_table\_iterator[i].oprnd\_ << endl;

}

cout << endl;

resultfile.close();

}

else cout << "Syntax Error" << endl;

}