C++ SIMPLE COMPILER

Hunter Damron and Brennan Cain

main.cpp

/\*

\* Main class calls compiler.cpp to compile SIMPLE to SML

\* File: src/main.cpp

\* Authors: Hunter Damron and Brennan Cain

\* (hdamron17) (brenn10)

\*/

#include <iostream>

#include <fstream>

#include <vector>

#include <sstream>

#include "compiler.h"

using namespace std;

int main(int argc, char\*\* argv) {

if(argc == 2)

{

compiler::compile(argv[1], "out.sml");

}

else if(argc == 3)

{

compiler::compile(argv[1], argv[2]);

}

else

{

cerr << "You must provide a SIMPLE source file\n";

}

}

compiler.h

/\*

\* Header for compiler.cpp class which compiles SIMPLE to SML

\* File: src/compiler.h

\* Authors: Hunter Damron and Brennan Cain

\* (hdamron17) (brenn10)

\*

\* Created on October 24, 2016, 10:08 AM

\*/

#ifndef COMPILER\_H

#define COMPILER\_H

#include <fstream>

#include <iostream>

#include <vector>

#include <unordered\_set>

#include <unordered\_map>

#include <map>

class compiler {

public:

static std::vector<std::string> tokenize(std::string, std::string); //tokenizes string into a vector

static std::string replace\_all(std::string, std::string, std::string); //search and replace all in string

static std::string fmt(std::string, int size, char fill); //formats string with leading characters

static int manual\_stoi(std::string); //does stoi, but in a predictable manner

static std::string manual\_to\_string(int num); //manual to\_string

static void compile(std::string, std::string); //compiles from one file into another

static bool precheck(std::vector<std::string>); //prechecks postfix conversion

private:

static std::string compile(std::istream\*); //converter method makes a compiler instance and compiles

explicit compiler();

virtual ~compiler();

std::string get\_sml(std::istream\*);

std::vector<std::vector<std::string>> parse(std::istream\*);

std::string make\_sml(std::vector<std::vector<std::string>>\*);

std::string input(std::vector<std::string>\*);

std::string output(std::vector<std::string>\*);

std::tuple<std::string,int> let(std::vector<std::string>\*);

std::string \_goto(std::vector<std::string>\*);

std::string \_if(std::vector<std::string>\*);

std::string second\_parse(std::string);

int precedence(std::string, std::string); //TODO could be const but would not compile

std::vector<std::string> to\_postfix(std::vector<std::string>); //TODO same as above about const

std::vector<std::vector<std::string>> simple; //2D vector of simple code

std::unordered\_set<int> addresses; //Map of SIMPLE code addresses to SML addresses

//Note: needed addresses are written in the form A42 for address 42

std::unordered\_set<std::string> vars;

//Note: all vars are single letters in SIMPLE code, but others are allowed

std::unordered\_set<std::string> initialized\_vars;

//initialized vars must include all vars by the end

std::unordered\_set<int> constants;

//Note: constants are written in the form A42 for number 42

std::unordered\_map<int,int> address\_map;

//Note: address map has actual addresses mapped SIMPLE to SML

int stack\_size = 0; //Number of stack memory locations necessary as S##

int program\_size = 0; //SML physical position of last command + 1

};

#endif /\* COMPILER\_H \*/

compiler.cpp

/\*

\* Compiler class parses SIMPLE code and outputs SML code

\* File: src/compiler.cpp

\* Authors: Hunter Damron and Brennan Cain

\* (hdamron17) (brenn10)

\* Created on October 24, 2016, 10:08 AM

\*/

#include <fstream>

#include <vector>

#include <stack>

#include <sstream>

#include <iostream>

#include <limits.h>

#include <cstdlib>

#include <string>

#include "compiler.h"

using namespace std;

//For testing if valid variable name

static const string ALPHA = "abcdefghijklmnopqrstuvwxyz";

/\*\*

\* Takes two file names, create input/out put streams, calls hidden constructor, closes streams

\*

\* @param infile Input file name

\* @param outfile Output file name

\*

\* @return void

\*/

void compiler::compile(string infile, string outfile)

{

//open istream from infile

ifstream\* in = new ifstream(infile, ios::in);

if(!(\*in))

{

cerr << "Invalid input file\n";

exit(EXIT\_FAILURE);

}

//call compiler

string sml = compile(in);

//delete input stream

delete in;

//open ostream from ofile

ofstream\* out = new ofstream(outfile,ios::out);

if(!(\*out))

{

cerr << "Invalid output file\n";

exit(EXIT\_FAILURE);

}

//output to file

(\*out) << sml;

//close output stream and delete

out->close();

delete out;

}

/\*\*

\* Takes input stream, runs through the interpreter, and returns SML string

\*

\* @param \*in Input stream

\*

\* @return Returns a string containing complete SML code

\*/

string compiler::compile(istream \*in)

{

//create the compiler as an object

compiler cpl;

//compile and save to string

string sml = cpl.get\_sml(in);

//push to out

return sml;

}

/\*\*

\* Takes an input stream and calls both parses on it converting to SML

\*

\* @param \*in Input Stream

\*

\* @return SML code as a string

\*/

string compiler::get\_sml(istream \*in)

{

vector<vector<string>> input = parse(in);

string pseudoSML = make\_sml(&input);

string realSML = second\_parse(pseudoSML);

return realSML;

}

/\*\*

\* Prevents external instantiation

\*/

compiler::compiler() {}

/\*\*

\* Default destructor

\*/

compiler::~compiler() {}

/\*\*

\* Parses data from istream into 2D vector for processing

\*

\* @param input Input istream with SIMPLE code

\*

\* @return Returns 2D vector with each statement on a row and each word a term

\*/

vector<vector<string>> compiler::parse(istream \*input)

{

//declare return variable

vector<vector<string>> parsed;

//for each line

while(!input->eof())

{

string in;

//Get the line

getline((\*input),in);

#if \_\_linux\_\_ || \_\_APPLE\_\_

//fix annoying line encoding issue

if(in.size() >= 1 && in[in.size()-1] == '\r')

{

in.erase(in.size()-1, 1); //remove last character if \r because we ain't no Windoze users

}

#elif \_WIN32

if(in.size() >= 2 && in[in.size()-2] != '\r' && in[in.size()-1] == '\n')

{

in.insert(in.size()-1, "\r"); //insert \r before ending \n

}

#else

#error Platform not supported

#endif

//split and save to a vector

vector<string> line = tokenize(in," ");

//push the vector to the return variable

parsed.push\_back(line);

}

return parsed;

}

/\*\*

\* Generates 2D SML code vector from 2D SIMPLE code vector

\*

\* @param simple\_code 2D vector containing SIMPLE code (produced by parse())

\*

\* @return Returns string containing SML code in same format as simple\_code

\*/

string compiler::make\_sml(vector<vector<string>> \*simple\_code)

{

stringstream sml\_stream;

int line\_tracker = -1; //keeps track of line numbers to maintain increasing order

//go through each line of the program

for(vector<string> line : (\*simple\_code) )

{

//If the line is valid

if(line.size() >= 2)

{

try //convert linenum as a string to a int

{

int linenum = manual\_stoi(line[0]);

if(linenum>99 or linenum<0)

{

cerr<< "Program too large on line " << linenum << endl;

exit(EXIT\_FAILURE);

}

if(linenum < line\_tracker)

{

cerr << "Line number out of order at line " << linenum << endl;

exit(EXIT\_FAILURE);

} else if(linenum == line\_tracker) {

cerr << "Duplicate line number " << linenum << endl;

exit(EXIT\_FAILURE);

}

if(address\_map.count(linenum) > 0)

{

cerr << "Duplicate line number " << linenum << endl;

exit(EXIT\_FAILURE);

}

address\_map.insert({linenum, program\_size});

line\_tracker = linenum;

}

catch(invalid\_argument& e) //bad linenum

{

cerr<<"Invalid line number " << line[0]<<endl;

exit(EXIT\_FAILURE);

}

catch(exception e)//even worse linenum

{

cerr<<"Line number too large at "<< line[0] << endl;

exit(EXIT\_FAILURE);

}

//get the basic command

string command = line[1];

if(command == "rem") // rem

{

//TODO nothing

}

else if(command == "input") //input

{

sml\_stream << input(&line);

program\_size++;

}

else if(command == "print") //print

{

sml\_stream << output(&line);

program\_size++;

}

else if(command == "goto") //goto

{

sml\_stream << \_goto(&line);

program\_size++;

}

else if(command == "if") // if

{

sml\_stream << \_if(&line);

}

else if(command == "let") // let

{

tuple<string,int> cmd = let(&line);

sml\_stream << get<0>(cmd);

program\_size += get<1>(cmd);

}

else if(command == "end") //DIE!!! (actually finish nicely)

{

sml\_stream << "4300\n";

program\_size++;

}

else //invalid command

{

cerr << "Command \"" << line[1] << "\" invalid."<<endl;

exit(EXIT\_FAILURE);

}

}

else if(line.size() == 0)

{

//Do nothing on empty line

}

else //line is invalid

{

cerr << "Invalid line " << line.at(0) << endl;

exit(EXIT\_FAILURE);

}

}

return sml\_stream.str();

}

/\*\*

\* Writes SML for input command, putting filler for variable address

\*

\* @param cmd Tokenized SIMPLE input command (<linenum> input <var>)

\*

\* @return Returs full SML string (always a single line for input command)

\*/

string compiler::input(vector<string> \*cmd)

{

//Check that the line is the correct length for a input "# input id"

if(cmd->size() == 3)

{

//get the id

string var = cmd->at(2);

initialized\_vars.insert(var); //add to initialized vars

//check if id or constant

if(ALPHA.find(var) != string::npos)

{

vars.insert(var);

}

else //if not a valid id

{

cerr << "Invalid id \"" << var << "\" on line " << cmd->at(0) << endl;

exit(EXIT\_FAILURE);

}

stringstream sml;

//create the sml code

sml << "10" << var << endl;

return sml.str();

}

else //invalid command

{

cerr<<"Invalid argument to input on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

}

/\*\*

\* Writes SML for outpt command, putting filler for variable address

\*

\* @param cmd Tokenized SIMPLE output command (<linenum> output <var>)

\*

\* @return Returs full SML string (always a single line for output command)

\*/

string compiler::output(vector<string> \*cmd)

{

//Check if valid command length

if(cmd->size() == 3)

{

//get the id to output

string var = cmd->at(2);

//check if valid id

if(ALPHA.find(var) != string::npos)

{

vars.insert(var);

}

else //if not a valid id

{

cerr<<"Invalid variable \"" << var << "\" on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

//create the sml for the output

stringstream sml;

sml << "11" << var << endl;

return sml.str();

}

else //BAD JUJU

{

cerr<<"Invalid command format on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

}

/\*\*

\* Writes SML for output command, putting filler for variable address

\*

\* @param cmd Tokenized SIMPLE output command (<linenum> output <var>)

\*

\* @return Returns full SML string (always a single line for output command)

\*/

string compiler::\_if(vector<string> \*cmd)

{

//check that the comman is in a valid format

if(cmd->size()==7)

{

//Get id1

string id1 = cmd->at(2);

if(ALPHA.find(id1) != string::npos) //check if a valid id

{

vars.insert(id1);

}

else //If not valid, die

{

cerr<<"Invalid variable \"" << id1 << "\" on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

//get id2

string id2 = cmd->at(4);

if(ALPHA.find(id2) != string::npos) //check if valid id

{

vars.insert(id2);

}

else //if not valid, die

{

cerr<<"Invalid variable \"" << id2 << "\" on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

//get the address

int address;

try //get address if valid

{

address = manual\_stoi(cmd->at(6));

if(address>99 or address<0)//address out of bounds, die

{

cerr<<"Goto address out of bounds " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

}

catch(invalid\_argument& e) //bad address, die

{

cerr<<"Invalid goto address on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

addresses.insert(address);

/\*

\*Add ye ol' sml to the code and increment the program size

\*/

stringstream sml;

string relop = cmd->at(3); // get the relop from the line

/\*

\* This is a bunch of if statements for each relop

\* the SML is similar but loads the ids in different

\* order and uses different branching

\*/

if(relop == "==")

{

sml << "20" << id1 << endl;

sml << "31" << id2 << endl;

sml << "42A" << address << endl;

program\_size += 3;

}

else if(relop == "!=")

{

sml << "20" << id1 << endl;

sml << "31" << id2 << endl;

sml << "42" << program\_size+5 << endl;

sml << "40A" << address<<endl;

program\_size += 4;

}

else if(relop == ">=")

{

sml << "20" << id2 << endl;

sml << "31" << id1 << endl;

sml << "41A" << address << endl;

sml << "42A" << address << endl;

program\_size += 4;

}

else if(relop == "<=")

{

sml << "20" << id1 << endl;

sml << "31" << id2 << endl;

sml << "41A" << address << endl;

sml << "42A" << address << endl;

program\_size += 4;

}

else if(relop == ">")

{

sml << "20" << id2 << endl;

sml << "31" << id1 << endl;

sml << "41A" << address<< endl;

program\_size += 3;

}

else if(relop == "<")

{

sml << "20" << id1 << endl;

sml << "31" << id2 << endl;

sml << "41A" << address<< endl;

program\_size += 3;

}

else //the relop was not specified or specified incorrectly, thus we DIE (like hunter will if he keeps doing todos)

{

cerr << "Relop issues on line " << cmd->at(0) << ". I had to kill myself because of YOU!"<<endl;

exit(EXIT\_FAILURE);

}

return sml.str();

}

else

{

cerr << "Relop issues on line " << cmd->at(0) << ". I had to kill myself because of YOU!"<<endl;

exit(EXIT\_FAILURE);

}

}

/\*\*

\* Writes the SML for a goto statement

\*

\* @param cmd the pointer to the line of a comand

\*

\* @return sml as a string

\*/

string compiler::\_goto(vector<string> \*cmd)

{

if(cmd->size() == 3) //check that the size of the command ifs valid

{

int linenum;

try //attempt to get ye ol lien num

{

linenum = manual\_stoi(cmd->at(2));

if(linenum>99 or linenum<0) //out of bounds line number

{

cerr<<"Line pointer out of bounds on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

addresses.insert(linenum);

}

catch(invalid\_argument& e) // bad linenum

{

cerr<<"Invalid line number on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

//save the command to sml

stringstream sml;

sml << "40A" << linenum << endl; //puts temporary value starting with 'a'

string ret = sml.str();

return ret;

}

else //illegal number of arguments to the thingy

{

cerr<<"Invalid number of arguments on line " << cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

}

/\*\*

\* Checks if infix math is valid

\*

\* @param infix Tokenized vector of infix math

\*

\* @return Returns true if everything works, else false

\*/

bool compiler::precheck(vector<string> infix)

{

string ops = "+-/\*";

string paren="()";

if(infix.size()==1 and ops.find(infix.at(0))==string::npos and paren.find(infix.at(0))==string::npos)

{

return true;

}

for(int i = 0; i < infix.size()-1;i++)

{

if(ops.find(infix.at(i))!=string::npos and ops.find(infix.at(i+1))!=string::npos)

{

cerr << "Double operator in infix" ;

return false;

}

if(infix.at(i)=="(" and ops.find(infix.at(i+1))!=string::npos)

{

cerr << "Open parenthesis followed by operator";

return false;

}

if(ops.find(infix.at(i))!=string::npos and infix.at(i+1)==")")

{

cerr << "Operator followed by close parenthesis";

return false;

}

if(infix.at(i)=="(" and infix.at(i+1)==")")

{

cerr << "Nonsensical open and close parentheses. Y U DO DIS.";

return false;

}

if(infix.at(i)==")" and infix.at(i+1)=="(")

{

cerr << "Close and open parentheses with no meat in between. Where's the beef?!";

return false;

}

if(not (ops.find(infix.at(i))!=string::npos or paren.find(infix.at(i))!=string::npos) and not (ops.find(infix.at(i+1))!=string::npos or paren.find(infix.at(i+1))!=string::npos))

{

cerr << "Two ids/constants next to each other." ;

return false;

}

if(infix.at(i)=="(")

{

if(i!=0)

{

if(ops.find(infix.at(i-1))==string::npos and infix.at(i-1)!="(")

{

cerr<<"Open parenthesis prepended by non operator/non parenthesis" ;

return false;

}

}

}

if(infix.at(i)==")")

{

if(ops.find(infix.at(i+1))!=string::npos and infix.at(i+1)!=")")

{

cerr<<"Open parenthesis prepended by non operator/non parenthesis" ;

return false;

}

}

}

int parens=0;

for(auto i : infix)

{

if(i=="(") parens++;

else if(i==")") parens--;

}

if(parens!=0)

{

cerr << "Invalid parenthetication";

return false;

}

return true;

}

/\*\*

\* Creates the SML for the let command

\*

\* @param \*cmd pointer to the line with the let command

\*

\* @return tuple(sml code, size of command)

\*/

tuple<string,int> compiler::let(vector<string> \*cmd) {

if(cmd->size() >= 5 && cmd->at(3) == "=") {

string final\_var = "";

if(ALPHA.find(cmd->at(2)) != string::npos) {

final\_var = cmd->at(2); //final variable to hold value

initialized\_vars.insert(final\_var); //add final var to initialized vars

}

else

{

cerr << "Invalid variable on line " << cmd->at(0) << endl;

exit(EXIT\_FAILURE);

}

vector<string> infix(cmd->begin()+4, cmd->end()); //with only math part

bool good\_code = precheck(infix); //check validity

if(!good\_code)

{

cerr << " on line " << cmd->at(0) << endl;

}

vector<string> postfix = to\_postfix(infix); //math part to postfix

stringstream sml;

string operators="+-/\*"; /\*\*/

stack<string> ids;

int let\_size = 0;

int local\_stack = 0;

if(cmd->size() == 5) //only one value (i.e. "10 let x = 9" )

{

stringstream sml;

string token = cmd->at(4); //value to be stored in final var

if(ALPHA.find(token) != string::npos) { //valid variable

sml << "20" << token << endl; //load variable into acc for moving

vars.insert(token); //add variable to vars list

}

else // now it may be a literal but we'll see about that

{

int num = 0;

try

{

num = manual\_stoi(token);

}

catch(invalid\_argument& e)

{

cerr << "Invalid Argument on line " <<cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

catch(out\_of\_range& e)

{

cerr << "Big thing on line " <<cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

sml << "20C" << manual\_to\_string(num) << endl; //load constant to acc for moving

constants.insert(num); //add constant to constants list

}

sml << "21" << final\_var << endl;

return make\_tuple(sml.str(), 2);

}

else if(cmd->size() == 6)

{

cerr << "Invalid number of parameters in let on line " <<cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

else

{

for(auto i = postfix.begin(); i < postfix.end(); i++) {

string token = (\*i);

if(operators.find(token) != string::npos) { //pointing to operator

string id2 = ids.top();

ids.pop();

string id1 = ids.top();

ids.pop();

if(id1[0] == 'S') //if it starts with S, it's popped off stack

{

local\_stack--;

}

if(id2[0] == 'S') //if it starts with S, it's popped off stack

{

local\_stack--;

}

int operation = 30 + operators.find(token);

//uses position plus 30 to convert to SML opcode

sml << "20" << id1 << endl //loads id1 into accumulator

<< operation << id2 << endl; //applies operator on acc with id2 param

if(i + 1 == postfix.end()) //last operator so save to final var

{

sml << 21 << final\_var << endl; //store to final var

}

else

{

sml << 21 << "S" << local\_stack << endl; //else store to stack

ids.push("S" + manual\_to\_string(local\_stack)); //push the s to local stack

local\_stack++; //makes stack 1 larger

}

let\_size += 3; //each operator uses 3 SML ops

if(local\_stack + 1 > stack\_size) //this requires more stack

stack\_size = local\_stack;

} else { //hopefully pointing to constant or variable

if(ALPHA.find(token) != string::npos) { //valid variable

ids.push(token);

vars.insert(token);

} else { // now it may be a literal but we'll see about that

int num = 0;

try {

num = manual\_stoi(token);

} catch(invalid\_argument& e) {

cerr << "Invalid Argument on line " <<cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

} catch(out\_of\_range& e) {

cerr << "Big thing on line " <<cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

ids.push("C" + manual\_to\_string(num));

constants.insert(num);

}

}

}

return make\_tuple(sml.str(), let\_size);

}

} else {

cerr << "Invalid command format on line "<<cmd->at(0)<<endl;

exit(EXIT\_FAILURE);

}

}

/\*\*

\* Replaces temporary variable names with physical addresses for final SML code

\* -Also checks that SML code fits within 100 op limit

\*

\* @param partial\_sml SML code containing variable names

\*

\* @return Returns complete SML code with all variable names replaced

\*/

string compiler::second\_parse(string partial\_sml) {

if(program\_size + stack\_size + vars.size() + constants.size() > 100) {

cerr << "Program too large." << endl;

}

//Replace addresses

for(auto iter = addresses.begin(); iter != addresses.end(); iter++) {

try {

int replace = address\_map.at(\*iter);

string newstr = fmt(manual\_to\_string(replace), 2, '0');

partial\_sml = replace\_all(partial\_sml, "A" + manual\_to\_string(\*iter), newstr);

} catch (out\_of\_range& e) {

cerr << "Invalid reference to line number " << \*iter << endl;

exit(EXIT\_FAILURE);

}

}

//Replace constants

for(auto iter = constants.begin(); iter != constants.end(); iter++) {

string newstr = fmt(manual\_to\_string(program\_size), 2, '0');

partial\_sml = replace\_all(partial\_sml, "C" + manual\_to\_string(\*iter), newstr)

+ fmt(manual\_to\_string(\*iter), 4, '0') + "\n";

program\_size++;

}

//Replace stack variables

for(int i = 0; i < stack\_size; i++) {

string newstr = fmt(manual\_to\_string(program\_size), 2, '0');

partial\_sml = replace\_all(partial\_sml, "S" + manual\_to\_string(i), newstr) + "0000\n";

program\_size++;

}

//Replace variables and check if initialized

for(string var : vars) {

if(initialized\_vars.find(var) == initialized\_vars.end())

{

cerr << "Variable " << var << " is never initialized" << endl;

exit(EXIT\_FAILURE);

}

string newstr = fmt(manual\_to\_string(program\_size), 2, '0');

partial\_sml = replace\_all(partial\_sml, var, newstr) + "0000\n";

program\_size++;

}

return partial\_sml;

}

/\*\*

\* Replaces all instances of the old string to the new string in str

\*

\* @param str String in which replacements will be made

\* @param oldstr String to be replaced

\* @param newstr String to be put in place of oldstr

\*

\* @return Returns str but with replacements made

\*/

string compiler::replace\_all(string str, string oldstr, string newstr)

{

int oldsize = oldstr.size();

int strsize = str.size();

stringstream ret;

int prev = 0;

for(int i = 0; i + oldsize <= strsize; i++)

{

if(oldstr == str.substr(i, oldsize))

{

ret << str.substr(prev, i-prev) << newstr;

i += oldsize;

prev = i;

}

}

ret << str.substr(prev, strsize-1);

return ret.str();

}

/\*\*

\* Formats the string to output with leading zeroes

\*

\* @param original original string

\* @param size how large the string should be at the end

\* @param fill char to fill empty with

\*

\* @return string of string with leading zeroes

\*/

string compiler::fmt(string original, int size, char fill) {

int origsize = original.size();

if(origsize < size) {

for(int i = 0; i < size - origsize; i++) {

original = fill + original;

}

}

return original;

}

/\*\*

\* Gives which op is of higher precedence

\*

\* @param op1 first operation

\* @param op2 second operation

\*

\* @return int 1: first is higher, 0: both are equal, -1: second is higher

\*

\* @author Brennan Cain

\*/

int compiler::precedence(string op1, string op2)

{

string t1="^";

string t2="\*/";

string t3="+-";

if( (t1.find(op1)!=string::npos and t1.find(op2)!=string::npos) or \

(t2.find(op1)!=string::npos and t2.find(op2)!=string::npos) or \

(t3.find(op1)!=string::npos and t3.find(op2)!=string::npos))

{

return 0;

}

else if(t1.find(op1)!=string::npos)

{

return 1;

}

else if(t1.find(op2)!=string::npos)

{

return -1;

}

else if(t2.find(op1)!=string::npos)

{

return 1;

}

else if(t2.find(op2)!=string::npos)

{

return -1;

}

else

{

return 1;

}

}

/\*\*

\* Converts a vector of infix to a queue of postfix

\* @param infix vector<string> golds the infix string broken on spaces

\*

\* @return queue<string> postfixed statement

\*

\* @author Brennan Cain

\*/

vector<string> compiler::to\_postfix(vector<string> infix)

{

string operators="-\*/+()";

vector<string> post;

stack<string> ops;

for(auto i : infix)

{

if(operators.find(i)==string::npos)//If a constant

{

post.push\_back(i);

}

else if(ops.empty())//If the op stack is empty, add the op

{

ops.push(i);

}

else if(ops.top()=="(" or i=="(")//If the new op or top op are left paren, add new op

{

ops.push(i);

}

else if(i==")")//If a right paren, pop ops until left paren

{

while(ops.top()!="(")

{

post.push\_back(ops.top());

ops.pop();

}

ops.pop();

}

else if(precedence(i,ops.top())==1)//if new op has higher precedence, push it

{

ops.push(i);

}

else if(precedence(i,ops.top())==0)//if the new op has equal precedence, pop and push new

{

post.push\_back(ops.top());

ops.pop();

ops.push(i);

}

else if(precedence(i,ops.top())==-1) //If new is lower, solve until new is equal or higher

{

while(precedence(i,ops.top())==-1)

{

post.push\_back(ops.top());

ops.pop();

if(ops.empty())//Prevents segmentation faults

{

break;

}

}

ops.push(i);

}

}

while(!ops.empty())//Flush the rest of the ops

{

post.push\_back(ops.top());

ops.pop();

}

return post;

}

/\*\*

\* Tokenizes one string into a vector of strings

\*

\* @param str String to be tokenized

\* @param delimiter String describing location of split between tokens

\*

\* @return Returns vector containing string tokens

\*

\* @author Hunter Damron

\*

\* @editor Brennan Cain ADDED THE NEWLINE BRACES

\*/

vector<string> compiler::tokenize(string str, string delimiter)

{

vector<string> tokens;

int start = 0;

int end = 0;

while (end != string::npos)

{

end = str.find(delimiter, start);

string sub = str.substr(start, end - start);

if (sub.size() > 0)

{

tokens.push\_back(sub);

}

start = end + delimiter.size();

}

return tokens;

}

/\*\*

\* Does stoi but manually (predictable unlike the real deal)

\*

\* @param str String to be parsed to int

\*

\* @return Returns string as an int

\*

\* @throws Throws out\_of\_range if number is too big or small

\* @throws Throws invalid\_argument if str does not have valid number

\*/

int compiler::manual\_stoi(string str)

{

bool negative = false; //true if str starts with negative sign

int value = 0; //value parsed

bool first = true; //true if first instance

for(char i : str)

{

if(first && str[0] == '-')

{

negative = false;

}

else

{

int digit = ((int)i) - 48; //Subtract 30 hex to be 0 - 10 if number

if(digit >= 0 && digit < 10)

{

if(value > INT\_MAX / 10) //multiplying by 10 will overflow

throw out\_of\_range("Number too big");

value \*= 10; //go up by power of 10

if(value > INT\_MAX - digit) //adding digit will overflow

throw out\_of\_range("Number too big");

value += digit; //add digit

}

else

{

throw invalid\_argument("Invalid character " + i);

}

}

first = false;

}

if(negative)

value = -value;

return value;

}

/\*\*

\* Manual manual\_to\_string(int) method

\*

\* @param num Number to be converted

\*

\* @return Returns string representation

\*/

string compiler::manual\_to\_string(int num) {

if(num == 0)

{

return "0";

}

stringstream out;

if(num < 0)

{

out << "-";

num = -num;

}

int power = 1;

while(power <= num)

power \*= 10;

while(power > 1)

{

out << (num % power) / (power / 10);

power /= 10;

}

return out.str();

}