(Print this page as a cover sheet for your printouts)

CSCE 420 HOMEWORK 2
Dr. Daugherity
Due: 11:59 P.M. Monday, October 30, 2017

"On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment."

_____ _____
Typed or printed name of student Signature of student

NOTE: Please follow your lab instructor's directions for submitting your assignment through CSNET. ONLY ASSIGNMENTS SUBMITTED TO CSNET WILL BE GRADED!
Make a printout of each source file and staple it behind this cover sheet. Sign it and turn it in in class Tuesday. IF YOU DO NOT TURN IN A SIGNED COVER
SHEET YOUR WORK WILL NOT BE GRADED!

NOTE: Homework will be graded on build.tamu.edu, using g++ 7.2.0 with -std=c++17, or javac and java, or python3.

You are free to develop your programs on any other platform, but it is your responsibility to make sure your programs also compile and execute correctly on
build.tamu.edu as specified.

NOTE: Each file submitted (hw2pr1.cpp, etc.--see below) must begin as follows:
```
//Your name and UIN
//CSCE 420
//Due: Ocotber 2, 2017
//hw2pr1.cpp (or whatever this file name is)
```

NOTE: Also write a README.txt file with whatever information is needed to compile and run your programs. Zip the README.txt and the homework files into
a single file named hw2.zip and submit to CSNET.

The grade for this lab will be based on style (formatting, variable names, comments, etc.), syntax (no compilation or link errors), and correctness (passes all test cases). Your grade for this lab is:

Problem #	1	2	3	4
Style	/2	/4	/4	/2
Syntax	/3	/6	/6	/3
Correctness	/5	/10	/10	/5
Total	/10	/20	/20	/10

Grand total _____/50

```
//Elliott Dobbs 823004322
//CSCE 420
//Due: October 30, 2017
//hw2pr1.cpp

#include <stdio.h>
#include <iostream>
#include <string>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <climits>
using namespace std;

struct Node{
    int value;
    Node * parent;
    vector<Node> subtrees;
};

//Function to remove whitespace from input
string removeWhite(string input){
    int i = 0;
    while (i < input.size()){
        if (input.substr(i, 1) == " ")
            input.erase(i, 1);
        else
            ++i;
    }
    return input;
}

//Function for creating a tree with a given space-less input tree
void createTree(string input, Node* current){

    //if the input is the last close parenthesis, we are at the root node
    //  with nothing left to do so we can exit
    if (input == ")")
        return;

    //Get the next character
    string symbol = input.substr(0, 1);
    input.erase(input.begin());

    //Actions based on what character is next
    // If this, create a new subtree and go to it (first location in
subtrees)
    if (symbol == "("){

        Node howdy;
        howdy.parent = current;
        howdy.value = (int)NULL;
        current->subtrees.push_back(howdy);
        createTree(input, &current->subtrees[0]);
```

```
    }
    //If this, create a subtree and go to it (last location in subtrees)
    else if (symbol == ","){

        Node howdy;
        howdy.parent = current;
        howdy.value = (int)NULL;
        current->subtrees.push_back(howdy);
        createTree(input, &current->subtrees.back());
    }
    //If this, go back to the parent node
    else if (symbol == ")"){

        createTree(input, current->parent);
    }
    //If this, get full number, make the node have that value,
    //  and then go to parent
    else{

        //Get the full number
        string test = input.substr(0, 1);
        while (test != "(" &&
               test != "," &&
               test != ")"){
            symbol.push_back(input[0]);
            input.erase(input.begin());
            test = input.substr(0, 1);
        }
        int leaf = stoi(symbol);
        current->value = leaf;

        createTree(input, current->parent);
    }
}

//Forward declaration for use in maxValue
void minValue(Node* n);

//Recursive call to get the maximum subtree value for the given node
void maxValue(Node* n){

    //If the number is -1, we need to get the value for it by
    //  calling minValue() for each subtree
    if (n->value == (int)NULL){

        int max = INT_MIN;
        for (int i = 0; i < n->subtrees.size(); ++i){

            minValue(&n->subtrees[i]);
            if (n->subtrees[i].value > max)
                max = n->subtrees[i].value;
        }

        n->value = max;
```

```
    }
}

//Recursive call to get the minimum subtree value for the given node
void minValue(Node* n){

    //If the number is -1, we need to get the value for it by
    //  calling maxValue() for each subtree
    if (n->value == (int)NULL){

        int min = INT_MAX;
        for (int i = 0; i < n->subtrees.size(); ++i){

            maxValue(&n->subtrees[i]);
            if (n->subtrees[i].value < min)
                min = n->subtrees[i].value;
        }

        n->value = min;
    }
}

int main(){

    string input;
    cout << "Enter a Tree:";
    cin >> input;
    input = removeWhite(input);

    Node root;
    root.parent = NULL;
    root.value = (int)NULL;

    createTree(input, &root);
    maxValue(&root);

    cout << endl << endl;
    cout << "The value of the root node is: " << root.value << endl << endl;

    return 1;

}
```

```
//Elliott Dobbs 823004322
//CSCE 420
//Due: October 30, 2017
//hw2pr2.cpp

#include <stdio.h>
#include <iostream>
#include <string>
#include <fstream>
#include <stdlib.h>
#include <vector>
#include <climits>
using namespace std;

struct Node{
    int value;
    Node * parent;
    vector<Node> subtrees;
};

//Function to remove whitespace from input
string removeWhite(string input){
    int i = 0;
    while (i < input.size()){
        if (input.substr(i, 1) == " ")
            input.erase(i, 1);
        else
            ++i;
    }
    return input;
}

//Function for creating a tree with a given space-less input tree
void createTree(string input, Node* current){

    //if the input is the last close parenthesis, we are at the root node
    //  with nothing left to do so we can exit
    if (input == ")")
        return;

    //Get the next character
    string symbol = input.substr(0, 1);
    input.erase(input.begin());

    //Actions based on what character is next
    // If this, create a new subtree and go to it (first location in
subtrees)
    if (symbol == "("){

        Node howdy;
        howdy.parent = current;
        howdy.value = (int)NULL;
        current->subtrees.push_back(howdy);
        createTree(input, &current->subtrees[0]);
```

```
    }
    //If this, create a subtree and go to it (last location in subtrees)
    else if (symbol == ","){

        Node howdy;
        howdy.parent = current;
        howdy.value = (int)NULL;
        current->subtrees.push_back(howdy);
        createTree(input, &current->subtrees.back());
    }
    //If this, go back to the parent node
    else if (symbol == ")"){

        createTree(input, current->parent);
    }
    //If this, get full number, make the node have that value,
    //  and then go to parent
    else{

        //Get the full number
        string test = input.substr(0, 1);
        while (test != "(" &&
               test != "," &&
               test != ")"){
            symbol.push_back(input[0]);
            input.erase(input.begin());
            test = input.substr(0, 1);
        }
        int leaf = stoi(symbol);
        current->value = leaf;

        createTree(input, current->parent);
    }
}

//Forward declaration for use in maxValue
void minValue(Node* n, int a, int b);

//Recursive call to get the maximum subtree value for the given node
void maxValue(Node* n, int a, int b){

    //If the number is NULL, we need to get the value for it by
    //  calling minValue() for each subtree
    if (n->value == (int)NULL){

        int max = INT_MIN;
        for (int i = 0; i < n->subtrees.size(); ++i){

            minValue(&n->subtrees[i], a, b);
            if (n->subtrees[i].value > max)
                max = n->subtrees[i].value;
```

```
            //If we a value in a  max level branch that is greater than beta,
then we do not need to search more since the max function will return this
value even though it is greater then what min will pick
            if (n->subtrees[i].value >= b){
                cout << "Beta Pruning..." << endl;
                break;
            }

            //Get a new alpha value if there is one (and reset beta)
            else{
                if (a >= n->subtrees[i].value)
                    continue;
                else
                    a = n->subtrees[i].value;
            }
        }

        n->value = max;
    }
}

//Recursive call to get the minimum subtree value for the given node
void minValue(Node* n, int a, int b){

    //If the number is NULL, we need to get the value for it by
    //  calling maxValue() for each subtree
    if (n->value == (int)NULL){

        int min = INT_MAX;
        for (int i = 0; i < n->subtrees.size(); ++i){

            maxValue(&n->subtrees[i], a, b);
            if (n->subtrees[i].value < min)
                min = n->subtrees[i].value;

            //If we reach a value in a min level branch that is less than
alpha, then we do not need to search more since the min function will return
this value even though it is less then what max will pick
            if (n->subtrees[i].value <= a){
                cout << "Alpha Pruning..." << endl;
                break;
            }

            //Get a new beta value if there is one
            else{
                if (b <= n->subtrees[i].value)
                    continue;
                else
                    b = n->subtrees[i].value;
            }
        }

        n->value = min;
    }
```

```
}

int main(){

    string input;
    cout << "Enter a Tree:";
    cin >> input;
    input = removeWhite(input);

    Node root;
    root.parent = NULL;
    root.value = (int)NULL;
    int alpha = INT_MIN, beta = INT_MAX;

    cout << endl << endl;

    createTree(input, &root);
    maxValue(&root, alpha, beta);

    cout << endl << endl;
    cout << "The value of the root node is: " << root.value << endl << endl;

    return 1;

}
```

```
//Elliott Dobbs 823004322
//CSCE 420
//Due: October 30, 2017
//hw2pr3.cpp

#include <stdio.h>
#include <iostream>
#include <string>
#include <fstream>
#include <stdlib.h>
#include <vector>
using namespace std;

//Symbol structure, value not used for this program
struct Symbol{
    string name;
    int value;
};

//Function that returns a list of symbols from a given PROLOG input string
vector<Symbol> getSymbols(string input){

    vector<Symbol> result;
    string tempString = "";
    char tempChar;
    int iter = 0, prevIter = 0;
    tempChar = input[iter];

    while (tempChar != '.'){

        prevIter = iter;

        //Iterate to the next invalid Symbol character
        while (tempChar != ':' && tempChar != ',' && tempChar != '.' &&
tempChar != '-'){
            ++iter;
            tempChar = input[iter];
        }

        //If prevIter and iter are the same, that means the next character
was
        //  also invalid and we need to loop again going to next character.
        //  If they are not the same, the string inside input between
location
        //  prevIter and iter is a symbol name and we can add it to the
result vector
        if (prevIter != iter){
            tempString = input.substr(prevIter, iter-prevIter);
            Symbol s;
            s.name = tempString;
            s.value = -1;
            result.push_back(s);
        }
        else{
```

```
            ++iter;
            tempChar = input[iter];
        }
    }

    return result;

}

//Function to get all possible deductions with the given clauses
vector<int> deduceSymbols(vector<Symbol> symbols,
                          vector<Symbol> queue,
                          vector< vector<int> > clauses,
                          vector<int> count,
                          vector<Symbol> dnq){

    //Go through each Symbol in the queue, decrementing the symbol count for
a clause
    //  if the symbol is found in the clause
    Symbol tempSymbol;
    int tempSymbolLocation;
    vector<int> result;

    while (queue.size() > 0){
        tempSymbol = queue[0];
        queue.erase(queue.begin());

        //getting the symbol location in the symbol vector
        for (int i = 0; i < symbols.size(); ++i){
            if (symbols[i].name == tempSymbol.name){
                tempSymbolLocation = i;
                break;
            }
        }

        //Check each clause for the symbol obtained from the queue
        for (int i = 0; i < clauses.size(); ++i){

            if (clauses[i].size() == 1)
                continue;
            else{
                for (int j = 1; j < clauses[i].size(); ++j){
                    if (clauses[i][j] == tempSymbolLocation){
                        --count[i];

                        //If we decrement a clause to 0, we can deduce the
first symbol
                        //  in the clause as true (as long as its not
declared false by
                        //  the dnq vector), and add it to the queue
                        if (count[i] == 0){
                            bool checkIfNDQ = false;
                            for (int k = 0; k < dnq.size(); ++k){
```

```
                                    if (dnq[k].name == symbols[clauses[i]
[0]].name){
                                        checkIfNDQ = true;
                                        break;
                                    }
                                }
                                if (!checkIfNDQ)
                                    queue.push_back(symbols[clauses[i][0]]);

                                bool alreadyDeducted = false;
                                for (int k = 0; k < result.size(); ++k){
                                    if (result[k] == clauses[i][0]){
                                        alreadyDeducted = true;
                                        break;
                                    }
                                }
                                if (!alreadyDeducted)
                                    result.push_back(clauses[i][0]);
                            }
                        }
                    }
                }
            }

    return result;
}

int main(){

    vector<Symbol> symbols;
    vector< vector<int> > clauses; /*Formatted as follows:
                                        -One vector for each clause
                                        -The first element in each clause
vector is either
                                            -a given true element if vector
size is 1
                                            -a element we can try to prove is
true by deduction
                                        -the rest of the elements are symbols
we need to be true
                                            in order to prove the first
element is true
                                        -Lastly, the elements are integers
representing the
                                            location of the symbol in the
symbol vector.
                                            (Serving as a kind of pointer)*/
    vector<int> count;
    vector<Symbol> queue;
    vector<Symbol> doNotQueue;

    //Getting input file
    ifstream myfile;
```

```
    string inputFile, inputLine;
    cout << "Enter a file to be used:";
    cin >> inputFile;
    myfile.open(inputFile);
    vector<Symbol> tempSymbols;
    bool foundFlag;

    //Each line of the file processed
    while(myfile >> inputLine){

        //Get list of symbols from this input line
        tempSymbols = getSymbols(inputLine);
        foundFlag = false;
        vector<int> tempVec;

        //Add the symbols to the symbol vector
        //If there are no symbols, simple add the whole first list of symbols
        //  to the symbols vector
        if (symbols.size() == 0){
            for (int i = 0; i < tempSymbols.size(); ++i){
                symbols.push_back(tempSymbols[i]);
                tempVec.push_back(i);
            }

            //Setting clauses vector
            clauses.push_back(tempVec);
        }

        //If there are already symbols, check for new symbols to add to the
symbol list
        else{
            for (int i = 0; i < tempSymbols.size(); ++i){
                for (int j = 0; j < symbols.size(); ++j){

                    if (tempSymbols[i].name == symbols[j].name){
                        foundFlag = true;
                        tempVec.push_back(j);
                    }
                }
                if (!foundFlag){
                    symbols.push_back(tempSymbols[i]);
                    tempVec.push_back(symbols.size() - 1);
                }
                else
                    foundFlag = false;
            }

            //Setting clauses vector
            clauses.push_back(tempVec);
        }

        //Add the symbol to queue if it is true, if it is declared as false,
make sure
        //  it is not ever added to the queue
```

```
        if (tempSymbols.size() == 1){
            if (inputLine.substr(0, 2) == ":-")
                doNotQueue.push_back(tempSymbols[0]);
            else
                queue.push_back(tempSymbols[0]);
        }

        //Add the number of symbols in right side of clause to the count
vector
        count.push_back(tempSymbols.size() - 1);

    }
    myfile.close();

    //Call the deduce function to deduce all symbols we can
    vector<int> deductedTrueSymbols = deduceSymbols(symbols, queue, clauses,
count, doNotQueue);

    //Print the output
    cout << endl;
    cout << "Deducted Symbols:" << endl << ": ";
    for (int i = 0; i < deductedTrueSymbols.size(); ++i){
        cout << symbols[deductedTrueSymbols[i]].name << " : ";
    }
    cout << endl << endl;

    return 1;
}
```