Answers to Practice Final Examination #2

Review session: Sunday, March 17, 3:00–5:00 P.M. (Hewlett 200)
Scheduled finals: Tuesday, March 19, 12:15–3:15 P.M. (Hewlett 200)
Thursday, March 21, 12:15–3:15 P.M. (Hewlett 200)

1. Simple algorithmic tracing (5 points)

```
          one
         /|
        /  |
   four  two
  / \
five nine
  / \
/   |
six  three
     /|
    /  |
   seven ten
```

2. Recursion (15 points)

```cpp
/*
 * Function: isReducible
 * Usage: if (isReducible(word, english)) . . .
 * --------------------------------------------------
 * Determines whether a word is reducible, which means that (a) it
 * is a word and (b) it is possible to delete some letter and still
 * have a reducible word. To simplify the recursion, the empty
 * string is defined to be reducible.
 */

bool isReducible(string word, Lexicon & english) {
    if (word.length() == 0) {
        return true;
    } else if (!english.contains(word)) {
        return false;
    } else {
        for (int i = 0; i < word.length(); i++) {
            if (isReducible(removeLetter(word, i), english)) return true;
        }
        return false;
    }
}

/*
 * Function: removeLetter
 * Usage: string shorter = removeLetter(str, k);
 * ---------------------------------------------
 * Returns the string of letters remaining after deleting the
 * character at index position k from the string str.
 */

string removeLetter(string str, int k) {
    if (k < 0 || k >= str.length()) {
        error("removeLetter: Index out of range");
    }
    str.erase(k, 1);
    return str;
}
```
3. Heap-stack diagrams and memory tracing (10 points)

Using explicit addresses:

Using pointers:
4. Linear structures and hash tables (15 points)

```cpp
/*
 * Function: listMultiwordMatches
 * Usage: listMultiwordMatches(words, webMap);
 * ---------------------------------------------------------------
 * Lists all of the matches for the sequence of words contained in the
 * first argument using the data for web matches stored in the webMap
 * table, as described in the problem.
 */

void listMultiwordMatches(Vector<string> & words,
                          Map< string,Vector<WebEntry> > & webMap) {
    foreach (WebEntry entry in webMap[words[0]]) {
        if (otherWordsMatch(words, entry, webMap)) {
            cout << entry.url << ":" << entry.index << endl;
        }
    }
}

/*
 * Function: otherWordsMatch
 * Usage: if (otherWordsMatch(words, entry, webMap)) ...
 * ---------------------------------------------------------------
 * Returns true if all the words (not counting the initial entry)
 * match the pattern specified by the entry structure. To do so,
 * those entries must have the same URL and an index that is
 * adjusted by the index of the word in the words list.
 */

bool otherWordsMatch(Vector<string> & words, WebEntry & entry,
                      Map< string,Vector<WebEntry> > & webMap) {
    WebEntry entryCopy = entry;
    for (int i = 1; i < words.size(); i++) {
        entryCopy.index++;
        if (!occursIn(entryCopy, webMap[words[i]])) return false;
    }
    return true;
}

/*
 * Function: occursIn
 * Usage: if (occursIn(entry, list)) ...
 * ---------------------------------------------------------------
 * Returns true if the entry appears somewhere in the list of matches.
 */

bool occursIn(WebEntry & entry, Vector<WebEntry> & list) {
    for (int i = 0; i < list.size(); i++) {
        if (entry.url == list[i].url && entry.index == list[i].index) {
            return true;
        }
    }
    return false;
}
```
5. Trees (15 points)

```c
/*
 * Implementation notes: unparse, unparseAtPrecedence
 * -------------------------------
 * The unparse function itself is a wrapper that uses unparseAtPrecedence
 * to do the real work. The prec argument indicates the current precedence.
 * Operators that have a lower precedence must be parenthesized. This
 * implementation ensures that operators associate to the left by
 * increasing the prevailing precedence for the right-hand operator
 * by 0.5; it would be equally effective to pass a boolean parameter
 * indicating whether this operand is a left or right child of its parent.
 */

string unparse(Expression *exp) {
    return unparseAtPrecedence(exp, 0);
}

string unparseAtPrecedence(Expression *exp, double prec) {
    ExpressionType type = exp->getType();
    if (type == CONSTANT || type == IDENTIFIER) {
        return exp->toString();
    } else {
        string op = exp->getOperator();
        Expression *lhs = exp->getLHS();
        Expression *rhs = exp->getRHS();
        int newPrec = precedence(op);
        string str = unparseAtPrecedence(lhs, newPrec);
        str += " " + op + " ";
        str += unparseAtPrecedence(rhs, newPrec + 0.5);
        if (newPrec < prec) str = "(" + str + ")";
        return str;
    }
}
```
6. Graphs (15 points)

```cpp
/*
 * Implementation notes: isBiconnected
 * ------------------------------
 * The only subtle piece of this implementation is the strategy
 * used in the inner foreach loop to select one node (first) and
 * put the other neighbors in a set (otherNeighbors). Once that
 * has been done, the solution to this problem is simply a matter
 * of checking that there is a path from first to each of the
 * other neighbors that doesn't require going through the node
 * you're removing from the graph.
 */

bool isBiconnected(Graph<Node, Arc> & g) {
    Set<Node *> visited;
    foreach (Node *node in g.getNodeSet()) {
        Set<Node *> neighbors = g.getNeighbors(node);
        Node *firstNeighbor = neighbors.first();
        foreach (Node *np in neighbors) {
            visited.clear();
            visited.add(node);
            if (!pathExistsVisited(firstNeighbor, np, visited)) {
                return false;
            }
        }
    }
    return true;
}

/* The pathExistsVisited function appears on Handout 52A */

bool pathExistsVisited(Node *start, Node *finish, Set<Node *> & visited) {
    if (visited.contains(start)) return false;
    visited.add(start);
    foreach (Arc *arc in start->arcs) {
        if (pathExistsVisited(arc->finish, finish, visited)) return true;
    }
    return false;
}
```

7. Data structure design (15 points)

7a)

```cpp
/* Private section */

class Grid {

    private:

    ValueType **elements; /* Two-level dynamic array of elements */
    int nRows; /* The number of rows in this grid */
    int nCols; /* The number of columns in this grid */

    public:

    // Constructor, Destructor, etc.

};
```
/* Implementation section */

/*
 * Notes on the representation
 * ---------------------------
 * The Grid is internally managed as a two-dimensional dynamic array of
 * elements.
 */

template <typename ValueType>
Grid<ValueType>::Grid(int numRows, int numCols) {
    if (numRows < 0 || numCols < 0) error("Illegal grid size");
    nRows = numRows;
    nCols = numCols;
    elements = new ValueType *[nRows];
    for (int i = 0; i < nRows; i++) {
        elements[i] = new ValueType[nCols];
    }
}

template <typename ValueType>
Grid<ValueType>::~Grid() {
    for (int i = 0; i < nRows; i++) {
        delete [] elements[i];
    }
    delete [] elements;
}

template <typename ValueType>
ValueType Grid<ValueType>::get(int row, int col) {
    if (!inBounds(row, col)) error("Grid index out of range");
    return elements[row][col];
}

template <typename ValueType>
void Grid<ValueType>::set(int row, int col, ValueType value) {
    if (!inBounds(row, col)) error("Grid index out of range");
    elements[row][col] = value;
}

template <typename ValueType>
bool Grid<ValueType>::inBounds(int row, int col) {
    return row >= 0 && col >= 0 && row < nRows && col < nCols;