CS106X Handout 35S
Autumn 2012 November 14th - 16th, 2012

Section Solution

Discussion Problem 1 Solution: Quadtrees

```
static quadtree *gridToQuadtree(Grid<bool>& image,
                                int lowx, int highx, int lowy, int highy) {
   quadtree *qt = new quadtree;
   qt->lowx = lowx; qt->highx = highx - 1;
   qt->lowy = lowy; qt->highy = highy - 1;
   if (allPixelsAreTheSameColor(image, lowx, highx, lowy, highy)) {
      qt->isBlack = image[lowx][lowy];
      for (int i = 0; i < 4; i++) {
         qt->children[i] = NULL;
      }
   } else {
      int midx = (highx - lowx) / 2;
      int midy = (highy - lowy) / 2;
      qt->children[NW] = gridToQuadtree(image, lowx, midx, midy, highy);
      qt->children[NE] = gridToQuadtree(image, midx, highx, midy, highy);
      qt->children[SE] = gridToQuadtree(image, midx, highx, lowy, midy);
      qt->children[SW] = gridToQuadtree(image, lowx, midx, lowy, midy);
      // assume NW, NE, etc are constants/#defines
   }

   return qt;
}

static quadtree *gridToQuadtree(Grid<bool>& image) {
   return gridToQuadtree(image, 0, image.numCols(), 0, image.numRows());
}
```

I don't bother with the code for **`allPixelsAreTheSameColor`**, because it's all kinds of obvious.

Discussion Problem 2 Solution: Patricia Trees

The **`containsWord`** routine is nontrivial, because it's as much about trees as it is about advanced string manipulation. It's complicated by the fact that the letters in the **`connection`** may be longer than the remaining portion of the word.

```
static int findConnection(const Vector<connection>& children,
                          const string& word) {
   for (int i = 0; i < children.size(); i++) {
      string prefix = word.substr(0, children.get(i).letters.size());
      int cmp = prefix.compare(children.get(i).letters);
      if (cmp == 0) return i;
      if (cmp < 0) break;
   }

   return -1;
}
```

```
static bool containsWord(const node *root, const string& word) {
   const node *curr = root;
   string clone = word;
   while (!clone.empty()) {
      int index = findConnection(curr->children, clone);
      if (index == -1) return false;
      clone = clone.substr(curr->children.get(index).letters.size());
      curr = curr->children.get(index).subtree;
   }

   return curr->isWord;
}
```

Discussion Problem 3 Solution: Regular Expressions

```
static void matchAllWords(const node *root, const string& regex,
                          Set<string>& matches, const string& workingPrefix) {

   if (root == NULL) return;
   if (regex.empty()) {
      if (root->isWord) {
         matches.add(workingPrefix);
      }
      return; // return regardless of whether the working prefix was a word
   }

   if (regex.size() == 1 || !ispunct(regex[1])) {
      matchAllWords(root->suffixes.get(regex[0]), regex.substr(1),
                    matches, workingPrefix + regex[0]);
   } else if (regex[1] == '?') {
      matchAllWords(root, regex.substr(2), matches, workingPrefix);
      matchAllWords(root->suffixes.get(regex[0]), regex.substr(2),
                    matches, workingPrefix + regex[0]);
   } else if (regex[1] == '*') {
      matchAllWords(root, regex.substr(2), matches, workingPrefix);
      matchAllWords(root->suffixes.get(regex[0]), regex,
                    matches, workingPrefix + regex[0]);
   } else { // assume regex[1] == '+'
      string equivregex = regex;
      equivregex[1] = '*'; // reframe y+ as yy*
      matchAllWords(root->suffixes.get(equivregex[0]), equivregex,
                    matches, workingPrefix + equivregex[0]);
   }
}

static void matchAllWords(const node *trie, const string& regex,
                          Set<string>& matches) {
   matchAllWords(trie, regex, matches, "");
}
```

Lab Problem 1 Solution: JavaScript Object Notation

Here's the core of my own parsing solution. I decided parsing arrays and dictionaries was complicated enough that I created functions to manage the details.

```
// prototype necessary for mutual recursion
static JSONElement *parseJSON(TokenScanner& s);
static Vector<JSONElement *> *parseJSONArray(TokenScanner& ts) {
  Vector<JSONElement *> *array = new Vector<JSONElement *>();
  bool firstElementConsumed = false;
  while (true) {
     string lookahead = ts.nextToken();
     if (lookahead == "]") return array;
     if (firstElementConsumed && lookahead != ",") {
        error("Oops!  Commas need to separate elements in a JSON array.");
     } else if (!firstElementConsumed) {
        ts.saveToken(lookahead);
     }

     JSONElement *element = parseJSON(ts);
     firstElementConsumed = true;
     array->add(element);
  }
}

static Map<string, JSONElement *> *parseJSONDictionary(TokenScanner& ts) {
  Map<string, JSONElement *> *dictionary = new Map<string, JSONElement *>();
  bool firstEntryConsumed = false;
  while (true) {
     string lookahead = ts.nextToken();
     if (lookahead == "}") return dictionary;
     if (firstEntryConsumed && lookahead != ",") {
        error("Oops!  JSON dictionary entries should be comma-delimited.");
     } else if (!firstEntryConsumed) {
        ts.saveToken(lookahead);
     }

     string key = ts.nextToken();
     if (ts.nextToken() != ":") {
        error("Keys and values should be colon-separated.");
     }

     JSONElement *value = parseJSON(ts);
     firstEntryConsumed = true;
     dictionary->put(key, value);
  }
}

static JSONElement *parseJSON(TokenScanner& ts) {
  string lookahead = ts.nextToken();
  if (lookahead.empty()) return NULL;
  JSONElement *element = new JSONElement;
  if (isdigit(lookahead[0])) {
     element->type = JSONElement::Integer;
     element->value.intValue = stringToInteger(lookahead);
  } else if (lookahead == "true" || lookahead == "false" ) {
     element->type = JSONElement::Boolean;
     element->value.boolValue = lookahead == "true";
  } else if (lookahead[0] == '"') {
     element->type = JSONElement::String;
```

```
      element->value.stringValue = new string(lookahead);
   } else if (lookahead == "[") {
      element->type = JSONElement::Array;
      element->value.arrayValue = parseJSONArray(ts);
   } else if (lookahead == "{") {
      element->type = JSONElement::Dictionary;
      element->value.dictionaryValue = parseJSONDictionary(ts);
   } else {
      error("JSON element type passed to parseJSON not recognized.");
   }

   return element;
 }
```

Printing is also complex enough for arrays and dictionaries that I went with some helper functions as well.

```
static void prettyPrintJSON(const JSONElement *jsonRoot); // forward prototype
static void prettyPrintJSONArray(const Vector<JSONElement *> *array) {
   cout << "[";
   bool firstEntryPrinted = false;
   for (int i = 0; i < array->size(); i++) {
      if (firstEntryPrinted) cout << ",";
      prettyPrintJSON(array->get(i));
      firstEntryPrinted = true;
   }
   cout << "]";
}

static void prettyPrintJSONDictionary(const Map<string, JSONElement *> *entries) {
   cout << "{";
   bool firstEntryPrinted = false;
   foreach (string key in (*entries)) {
      if (firstEntryPrinted) cout << ",";
      cout << key << ":";
      prettyPrintJSON(entries->get(key));
      firstEntryPrinted = true;
   }
   cout << "}";
}

static void prettyPrintJSON(const JSONElement *jsonRoot) {
   switch (jsonRoot->type) {
      case JSONElement::Integer:
         cout << jsonRoot->value.intValue;
         return;
      case JSONElement::Boolean:
         cout << (jsonRoot->value.boolValue ? "true" : "false");
         return;
      case JSONElement::String:
         cout << *(jsonRoot->value.stringValue);
         return;
      case JSONElement::Array:
         prettyPrintJSONArray(jsonRoot->value.arrayValue);
         return;
      case JSONElement::Dictionary:
         prettyPrintJSONDictionary(jsonRoot->value.dictionaryValue);
         return;
   }
}
```

Freeing memory is less complicated—uncomplicated enough that I didn't even bother writing helper functions.

```
static void disposeJSON(JSONElement *jsonRoot) {
   switch (jsonRoot->type) {
      case JSONElement::Integer:
      case JSONElement::Boolean:
         // value elements don't themselves need to be freed,
         break;
      case JSONElement::String:
         delete jsonRoot->value.stringValue;
         break;
      case JSONElement::Array:
         for (int i = 0; i < jsonRoot->value.arrayValue->size(); i++) {
            disposeJSON(jsonRoot->value.arrayValue->get(i));
         }
         delete jsonRoot->value.arrayValue;
         break;
      case JSONElement::Dictionary:
         foreach (string key in (*jsonRoot->value.dictionaryValue)) {
            disposeJSON(jsonRoot->value.dictionaryValue->get(key));
         }
         break;
   }

   delete jsonRoot;
}
```