JavaScript Object Notation Overview

JavaScript Object Notation, or JSON, is a popular, data exchange format that’s easily read by humans and easily processed by computers. It’s similar to HTML and XML in that all three encode a hierarchy of information, but JSON is visually easier to take, and it’s based on a small, simple subset of JavaScript.

For the purposes of this week’s section handout, we’re going to assume that the only primitive types of interest are Booleans, integers, and strings. We’ll pretend we’re in a world without decimal points, and characters can just be represented as strings of length one. We’ll further simplify everything so that:

- The only Boolean constants are `true` and `false`.
- All integers are nonnegative, so you’ll never encounter a negative sign. We’ll assume that all integers are small enough that they can fit into a C++ `int` without overflowing memory.
- Strings are delimited by double quotes (real JSON also allows strings to be delimited by single quotes as well, but we’ll pretend that’s not the case here).

Each of the following represents a legitimate JSON literal:

```
1 4124 4892014 true false "CS106X" "http://www.facebook.com" "hello there"
```

JSON arrays are way more interesting. The array is an ordered sequence much like the Vector, except that it’s heterogeneous and allows elements of varying types. Arrays are bookended by `"[" and "]"`, and commas are used to delimit the array elements.

Here are some simple array literals, where all elements are of the same type:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
[true, true, false, true]
["do", "re", "mi", "fa", "so", "la", "ti", "do"]
["Rachel", "Finn"], ["Brittany", "Artie"], ["Puck", "Quinn"]
```

Array entries aren’t required to be of the same type, so that peer elements needn’t be of the same type. That means these are all legit:

```
[1138, "star wars", false, [], [8, "C3PO", ["empire", [true], "R2D2"]]]
```
Problem 1: JavaScript Object Notation without Object Orientation

Assume the following type definition, which relies on something called a union. Your section leader will give you the lowdown on unions, but in a nutshell, they’re like structs, except you only use one of the fields at any one time.

```c
struct JSOElement {
    enum {
        Integer, String, Boolean, Array
    } type;
    union {
        int intValue;
        bool boolValue;
        string *stringValue;
        Vector<JSONElement *> *arrayValue;
    } value;
};
```

This JSOElement thing maintains state about a single value. The type field keeps tabs on whether the element is a Boolean, integer, string, or a composite array. The same type field is used to decide which of the four union fields should be used. The idea of creating a type tag to self-identify is used in many old school languages.

Below I’ve presented implementations for parse, print, and dispose to parse, print, and dispose of JSOElements for the three primitive types. You should extend the implementations of all three to work with JSON arrays, recognizing that the array elements can be integers, strings, Booleans, embedded arrays, or any combination of the four.

Assume the TokenScanner has been seeded with a legal JSON string, and that calls to nextToken will surface the sequence of tokens that make up that string. Here’s the code you’re starting with:

```c
static JSOElement *parse(TokenScanner& ts) {
    string lookahead = ts.nextToken();
    if (lookahead.empty()) return NULL;
    JSOElement *element = new JSOElement;
    if (isdigit(lookahead[0])) { // dispatch on token type via cascaded if/elses
        element->type = JSOElement::Integer;
        element->value.intValue = stringToInteger(lookahead);
    } else if (lookahead == "true" || lookahead == "false") {
        element->type = JSOElement::Boolean;
        element->value.boolValue = lookahead == "true";
    } else if (lookahead[0] == '"') {
        element->type = JSOElement::String;
        element->value.stringValue = new string(lookahead);
    } else { // TODO: add additional if statement to parse arrays
        error("JSON element type passed to parse not yet supported.");
    }
    return element;
}
```
static void print(const JSONElement *root) {
    switch (root->type) {
        case JSONElement::Integer:
            cout << root->value.intValue;
            return;
        case JSONElement::Boolean:
            cout << (root->value.boolValue ? "true" : "false");
            return;
        case JSONElement::String:
            cout << *(root->value.stringValue);
            return;
        case JSONElement::Array: // TODO: add code to handle arrays
            default:
                error("JSON element type passed to print not yet supported.");
    }
}

static void dispose(JSONElement *root) {
    switch (root->type) {
        case JSONElement::Integer:
        case JSONElement::Boolean:
            break;
        case JSONElement::String:
            delete root->value.stringValue;
            break;
        case JSONElement::Array: // TODO: add code to print arrays
            default:
                error("JSON element type passed to dispose not yet supported.");
    }
    delete root;
}

static void configureScanner(TokenScanner& ts, ifstream& infile) {
    ts.ignoreWhitespace();
    ts.scanStrings();
    ts.setInput(infile);
}

int main() {
    TokenScanner ts;
    ifstream infile("json-data.txt");
    configureScanner(ts, infile);
    while (true) {
        try {
            JSONElement *root = parse(ts);
            if (root == NULL) break;
            print(root);
            dispose(root);
            cout << endl;
        } catch (ErrorException& e) {
            cout << e.getMessage() << endl;
        }
    }
    cout << endl;
    cout << "Everything’s been read in and printed out." << endl;
    return 0;
};
Problem 2: JavaScript Object Notation with Object Orientation

Rewrite everything you wrote for Problem 1, this time modeling the JSONElement type hierarchy using objects, pure virtual methods, and inheritance instead of the OO-avoidant way using unions. The time spent working through Problem 1 was worth it, because that’s the approach you’d need to take in languages that don’t support object orientation and inheritance. By working through two different approaches, you’ll appreciate the elegance and cleanliness of the pure OO approach.

class JSONElement {
  public:
    virtual ~JSONElement() {};
    virtual string toString() const = 0;
  
  class JSONInteger : public JSONElement {
    public:
      JSONInteger(int i) { value = i; }
      string toString() const { return integerToString(value); }
    private:
      int value;
    
    // TODO: Add all other JSONElement subtypes and their implementations
    
    static JSONElement *parse(TokenScanner& scanner) {
      string lookahead = scanner.nextToken();
      if (lookahead.empty()) return NULL;
      if (isdigit(lookahead[0]))
        return new JSONInteger(stringToInteger(lookahead));
    // TODO: add several new lines to support other JSONElement subtypes
      error("JSON element type passed to parse not yet supported.");
      return NULL; // compiler can't tell it can never get here
    
    static void configureScanner(TokenScanner& ts, ifstream& infile) {
      ts.ignoreWhitespace();
      ts.scanStrings();
      ts.setInput(infile);
    
    int main() {
      TokenScanner ts;
      ifstream infile("json-data.txt");
      configureScanner(ts, infile);
      while (true) {
        try {
          JSONElement *root = parse(ts);
          if (root == NULL) break;
          cout << root->toString() << endl;
          delete root;
        } catch (ErrorException& ex) {
          cout << ex.getMessage() << endl;
        }
        cout << endl;
      }
      cout << "Everything's been read in and printed out." << endl;
      return 0;
    }
Problem 3: Muppet Inheritance

Consider the following set of class definitions (assume that all methods are public):

class Kermit {
    virtual void animal() = 0;
    void beaker() { muppet("Kermit::beaker"); animal(); }
    virtual void fozzie() { muppet("Kermit::fozzie"); rowlf(); }
    virtual void misspiggy() = 0;
    void rowlf() { muppet("Kermit::rowlf"); misspiggy(); }
    void muppet(string s) { cout << s << endl; }
};

class Statler : public Kermit {
    void beaker() { muppet("Statler::beaker"); rowlf(); }
    virtual void misspiggy() { muppet("Statler::misspiggy"); rowlf(); }
    void rowlf() { muppet("Statler::rowlf"); animal(); }
};

class Waldorf : public Statler {
    virtual void animal() { muppet("Waldorf::animal"); rowlf(); }
    void rowlf() { muppet("Waldorf::rowlf"); }
};

class Gonzo : public Kermit {
    virtual void animal() { muppet("Gonzo::animal"); rowlf(); }
    virtual void misspiggy() { muppet("Gonzo::misspiggy"); beaker(); }
    void rowlf() { muppet("Gonzo::rowlf"); }
};

Now consider the following function:

    void muppetShow(Kermit *kermit) {
        kermit->fozzie();
    }

What type of object can kermit legitimately address during execution? For each object type, list the output that would be produced by calling muppetShow against that type.