

Section Solution

Discussion Problem 1 Solution: Muppet Inheritance

The **Kermit** * can address either a **Waldorf** object or a **Gonzo** object. Each provides implementations of all of the methods—abstract or not—at the **Kermit** level. The **Kermit** class clearly has two abstract methods, and the **Statler** class doesn't provide an implementation for the **animal** method, so it's also an abstract class.

Output for **Waldorf** *

```
Kermit::fozzie  
Kermit::rowlf  
Statler::misspiggy  
Statler::rowlf  
Waldorf::animal  
Waldorf::rowlf
```

Output for **Gonzo** *

```
Kermit::fozzie  
Kermit::rowlf  
Gonzo::misspiggy  
Kermit::beaker  
Gonzo::animal  
Gonzo::rowlf
```

Discussion Problem 2 Solution: Scheme and Twisting Lists

```
(define twist (lambda (ls)  
  (if (or (null ls) (null (cdr ls))) ls  
      (cons (car (cdr ls)) (cons (car ls) (twist (cdr (cdr ls))))))))
```

Discussion Problem 3 Solution: Scheme and Sorting

```
(define issorted (lambda (ls)  
  (or (null ls)  
      (null (cdr ls))  
      (and (less (car ls) (car (cdr ls))) (issorted (cdr ls))))))
```

Lab Problem 1 Solution: JavaScript Object Notation Take II [code by Aubrey Gress]

```
class JSONElement {  
  public:  
    virtual ~JSONElement() {};  
    virtual string toString() = 0;  
};  
  
class JSONString : public JSONElement {  
  public:  
    JSONString(string str) { value = str; }  
    ~JSONString() {}  
    string toString() { return value; }  
  private:  
    string value;  
};  
  
class JSONInt : public JSONElement {  
  public:  
    JSONInt(int i) { value = i; }  
    ~JSONInt () {}  
};
```

```

    string toString() { return IntegerToString(value); };
private:
    int value;
};

class JSONBoolean : public JMLElement {
public:
    JSONBoolean(bool b) {value = b;}
    ~JSONBoolean () {}
    string toString() {return value ? "true" : "false"; };
private:
    bool value;
};

class JSONArray: public JMLElement {
public:
    JSONArray(Vector<JMElement*> arr) { value = arr; }
    ~JSONArray () {
        for (int i = 0; i < value.size(); i++)
            delete value[i];
    }
    string toString() {
        string str = "[";
        for (int i = 0; i < value.size(); i++) {
            str += value[i]->toString();
            if (i != value.size() - 1) {
                str += ", ";
            }
        }
        str += "]";
        return str;
    };
private:
    Vector<JMElement*> value;
};

class JSONDictionary: public JMLElement {
public:
    JSONDictionary(Map<JMElement*> dictionary) { value = dictionary; }
    ~JSONDictionary () {
        foreach(string key in value)
            delete value[key];
    }

    string toString() {
        string str = "{";
        int counter = 0;
        foreach(string key in value) {
            string keyToPrint = key;
            str += keyToPrint;
            str += " : ";
            str += value[key]->toString();
            if (counter != value.size() - 1) {
                str += ", ";
            }
            counter++;
        }
        str += "}";
        return str;
    };
private:

```

```

    Map<JSONElement*> value;
};

JSONElement *parseJSON(Scanner& s);

Vector<JSONElement *> parseJSONArray(Scanner& s) {
    Vector<JSONElement *> array;
    bool firstElementConsumed = false;
    while (true) {
        string lookahead = s.nextToken();
        if (lookahead == "]" ) return array;
        if (firstElementConsumed && lookahead != ",") {
            Error("Oops! Commas need to separate elements in a JSON array.");
        } else if (!firstElementConsumed) {
            s.saveToken(lookahead);
        }

        JSONElement *element = parseJSON(s);
        firstElementConsumed = true;
        array.add(element);
    }
}

Map<JSONElement *> parseJSONDictionary(Scanner& s) {
    Map<JSONElement *> dictionary;
    bool firstEntryConsumed = false;
    while (true) {
        string lookahead = s.nextToken();
        if (lookahead == "}") return dictionary;
        if (firstEntryConsumed && lookahead != ",") {
            Error("Oops! Commas need to separate entries in a JSON dictionary.");
        } else if (!firstEntryConsumed) {
            s.saveToken(lookahead);
        }

        string key = s.nextToken();
        if (s.nextToken() != ":") {
            Error("Expected a colon to separate the dictionary's key and value pair.");
        }

        JSONElement *value = parseJSON(s);
        firstEntryConsumed = true;
        dictionary.put(key, value);
    }
}

JSONElement *parseJSON(Scanner& s) {
    string lookahead = s.nextToken();
    if (lookahead.empty()) return NULL;

    if (isdigit(lookahead[0])) {
        return new JSONInt(StringToInteger(lookahead));
    } else if (lookahead == "true" || lookahead == "false" ) {
        return new JSONBoolean(lookahead == "true");
    } else if (lookahead[0] == '"') {
        return new JSONString(lookahead);
    } else if (lookahead[0] == '[') {
        return new JSONArray(parseJSONArray(s));
    } else if (lookahead[0] == '{') {
        return new JSONDictionary(parseJSONDictionary(s));
    } else {

```

```
        Error("JSON element type passed to parseJSON not yet supported.");
    }
    return NULL;
}

int main() {
    Scanner s;
    s.setSpaceOption(Scanner::IgnoreSpaces);
    s.setStringOption(Scanner::ScanQuotesAsStrings);

    ifstream infile("json-lite-data.txt");
    s.setInput(infile);

    while (true) {
        try {
            JMLElement *jsonRoot = parseJSON(s);
            if (jsonRoot == NULL) break;
            cout << jsonRoot->toString() << endl;
            delete jsonRoot;
            cout << endl;
        } catch (ErrorException& e) {
            cout << e.getMessage() << endl;
        }
    }

    cout << endl;
    cout << "Everything's been read in and printed out." << endl;
    return 0;
};
```