Problem 1: People You May Know

Because Facebook is interested in growing out its social graph, users are often presented with a list of other users you might be friends with even though that friendship isn’t officially recorded. That list is drawn from the set of Facebook users who are strictly two degrees away from you—that is, the list of your friends’ friends that aren’t already friends with you.

Assume that the following node definition is used to represent a Facebook user:

```c
struct user {
    int userID;       // unique
    string name;     // not necessarily unique
    Set<user *> friends; // assume friendship is symmetric
};
```

Write a function called `getFriendsOfFriends`, which given the address of your node in the social graph, returns as a `Set` the collection of nodes representing those on Facebook who are two degrees away from you.

```c
static Set<user *> getFriendsOfFriends(user *loggedinuser);
```

Problem 2: Detecting Cycles

Given access to a graph (in the form of the exposed `graph` introduced in lecture), write a predicate called `containsCycle`, which returns `true` if there are any cycles whatsoever in the graph, and `false` otherwise. The ability to detect cycles, and the ability to confirm that the addition of an edge doesn’t introduce cycles, is important for some applications (e.g. Stanford 1-2-3 needs to confirm that no two cell formulas mutually depend—directly or eventually—on each other, and C++ compilers sometimes elect to check that no two header files mutually `#include` one an other).

```c
static bool containsCycle(graph& g);
```
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 XML in that both 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 lowercase, and no delimiting quotes).
- 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 3: 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 JSOneElement {
    enum {
        Integer, String, Boolean, Array
    } type;
    union {
        int intValue;
        bool boolValue;
        string *stringValue;
        Vector<JSOneElement *> *arrayValue;
    } value;
};
```

This JSOneElement 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 JSOneElements 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.

Here’s the code you’re starting with:

```c
static JSOneElement *parse(TokenScanner& ts) {
    string lookahead = ts.nextToken();
    if (lookahead.empty()) return NULL;
    JSOneElement *element = new JSOneElement;
    if (isdigit(lookahead[0])) {
        element->type = JSOneElement::Integer;
        element->value.intValue = stringToInteger(lookahead);
    } else if (lookahead == "true" || lookahead == "false") {
        element->type = JSOneElement::Boolean;
        element->value.boolValue = lookahead == "true";
    } else if (lookahead[0] == '"') {
        element->type = JSOneElement::String;
        element->value.stringValue = new string(lookahead);
    } else {
        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;
    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;
    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."
    return 0;
};
Problem 4: JavaScript Object Notation with Object Orientation

Rewrite everything you wrote for Problem 3, this time modeling the JSONElement type hierarchy using objects, pure virtual methods, and inheritance instead of the OO-avoidant way using unions. The problem is precisely the same as for Problem 3, but the implementation is different and leverages your understanding of OOP and class design. I present a test program and enough of the hierarchy to manage numbers, and you should complete the hierarchy for Booleans, strings, and arrays.

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;  
};

static JSONElement *parse(TokenScanner& scanner) {  
    string lookahead = scanner.nextToken();  
    if (lookahead.empty()) return NULL;  
    if (isdigit(lookahead[0]))  
        return new JSONInteger(stringToInteger(lookahead));  
    error("JSON element type passed to parse not yet supported.");  
    return NULL; // compile 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;  
    }  
};