1 Introduction

JSON (JavaScript Object Notation) is a standard data interchange format. It supports the following basic data types:

- Number: A real number. It's represented in JSON as a decimal number, with an optional fractional part.
- String: A sequence of Unicode code points. It's represented in JSON by an opening quote ", followed by a sequence of characters, followed by a closing quote ". Special values, such as quotes, can be escaped with a backslash.
- Boolean: A boolean value. Represented by the text $true$ or $false$.
- Array: A sequence of JSON values. Represented by an opening bracket [, followed by a sequence of JSON values separated by a comma ,, followed by a closing bracket].
- Object: A map of strings to JSON values. Representing by an opening brace {, followed by a sequence of key/value pairs separated by a comma , followed by a closing brace }. A key/value pair is representedy by a string, followed by a colon :, followed by a JSON value.

2 Implementation

The `jsonEncode` function implements a basic encoding function. It acts as the entry-point into JSON encoding, and its only responsibility is to delegate to other functions based on the type of the argument. It takes an argument `val`, which is the value to encode. A JSON value is represented in memory as an array, where the first value is one of: "number", "string", "boolean", "array" or "object", and the second value is the object to be encoded.

```
\fun{jsonEncode}{val}{
    t := val.get(0);
    v := val.get(1);
    if t == "number" {
        return jsonEncodeNumber(v);
    } else if t == "string" {
        return jsonEncodeString(v);
    } else if t == "boolean" {
        return jsonEncodeBoolean(v);
    } else if t == "null" {
        return jsonEncodeNone(v);
    } else if t == "array" {
        return jsonEncodeArray(v);
    } else if t == "object" {
        return jsonEncodeObject(v);
    } else {
        return none;
    }
}
```

2.1 Encode Number

The jsonEncodeNumber function has to produce a string which contains the decimal expansion of a real number.

```
\fun{jsonEncodeNumber}{num}{
    return num.toString();
}
```

2.2 Encode String

The jsonEncodeString function has to produce a string which contains a JSON string literal, with proper escaping of quotes and backslashes.

```
\fun{jsonEncodeString}{str}{
    string := '"';
    index := 0;
    while index < str.length {
        ch := str.at(index);
        if ch == '\\' {
            string = string + '\\\\';
        } else if ch == '\"' {
            string = string + '\\"';
        } else {
            string = string + ch;
        }

        index = index + 1;
    }

    string = string + "\"";
    return string;
}
```

2.3 Encode Boolean

The `jsonEncodeBoolean` function needs to produce a string which contains either *true* or *false*.

```
\fun{jsonEncodeBoolean}{bool}{
    if bool {
        return "true";
    } else {
        return "false";
    }
}
```

2.4 Encode None

The `jsonEncodeNone` function needs to produce a string which contains *null*.

```
\fun{jsonEncodeNone}{val}{
    return "null";
}
```

2.5 Encode Array

The `jsonEncodeArray` function needs to produce an array of JSON values, which is represented by an opening bracket [, followed by comma-separated JSON values, followed by a closing bracket]. It uses the `jsonEncode` function to do all the heavy lifting of actually encoding the values.

```
\fun{jsonEncodeArray}{arr}{
    string := "[";
    index := 0;
    while index < arr.data.length {
        if index != 0 {
            string = string + ", ";
        }

        string = string + jsonEncode(arr.get(index));
        index = index + 1;
    }

    string = string + "]";
    return string;
}
```

2.6 Encode Object

The jsonEncodeObject function needs to produce a map of string keys to JSON values, which is represented by an opening brace {, followed by comma-separated key/value pairs, followed by a closing brace }. It uses the jsonEncode function to encode the values, and the jsonEncodeString function to encode the keys. It also uses the indent helper function to produce proper indentation.

A JSON object is represented in memory of an array of length $2l$, where l is the number of elements in the map. Elements at even indexes are keys, elements at odd indexes are values.

```
\fun{jsonEncodeObject}{map}{
    string := "{";
    index := 0;
    while index < map.data.length {
        if index != 0 {
            string = string + ", ";
        }

        key := map.get(index);
        val := map.get(index + 1);
        string = string + jsonEncodeString(key);
        string = string + ": ";
        string = string + jsonEncode(val);

        index = index + 2;
    }

    string = string + "}";
    return string;
}
```

3 Demonstration

Here's a simple example program which uses jsonEncode to encode a fairly complicated nested JSON structure.

```
\fun{main}{}{
    topLevel := Array();
    topLevel.push("exampleString");
    topLevel.push(jsonVal("string", "I'm a \"JSON string\""));

    topLevel.push("someNumber");
    topLevel.push(jsonVal("number", 100.57));
    topLevel.push("nothing");
    topLevel.push(jsonVal("null", none));

    exampleArray := Array();
    exampleArray.push(jsonVal("number", 100));

    nestedObject := Array();
    nestedObject.push("yes");
    nestedObject.push(jsonVal("boolean", "true"));

    nestedObject.push("no");
    nestedObject.push(jsonVal("boolean", false));

    exampleArray.push(jsonVal("object", nestedObject));

    topLevel.push("exampleArray");
    topLevel.push(jsonVal("array", exampleArray));

    print(jsonEncode(jsonVal("object", topLevel)));
}
```

The above code uses the `jsonVal` function, which returns a value tagged with the JSON type so that `jsonEncode` understands it. Here's the implementation:

```
\fun{jsonVal}{tag, val}{
    arr := Array();
    arr.push(tag);
    arr.push(val);
    return arr;
}
```

The output of that code should be:

```
{"exampleString": "I'm a \"JSON string\"", "someNumber": 100.57,
"nothing": null, "exampleArray": [100, {"yes": true, "no": false}]}
```