CS4530 Homework 2

Rocket Science

Context

AcmeRocket is working on software for in-flight trajectory adjustments based on thousands of on-board sensors. Accessing that many sensors takes time, to reduce latency, computations should be incremental. A colleague is working on exposing the sensors as files, and calling your code each time a sensor reading changes. Your job is to write the evaluator that computes incremental course corrections, for now the computations are written in a tiny language. Here's a sample program in that language.

```
a = watch("/var/s1")     # watch sensor s1, store in var a
b = a * 2                # define var b as double of a
c = watch("/var/s2")     # watch sensor s2
d = c + 1
e = b - a
f = watch("/var/s3")
g = f ?  e/f : d         # conditional assignment of e/f to g if f!=0
```

Variables hold floating point numbers. Variables can be defined only once. You only have to deal with "correct" programs, i.e. programs that use variables that have been defined before used and that do not divide by zero. At startup, a program is evaluated once with all sensor readings set to 0. Whenever a sensor is updated, the statements depending on it are evaluated again. Above, we have these dependencies

```
a depends on /var/s1
b depends on a
c depends on /var/s2
d depends on c
e depends on b and a
f depends on /var/s3
g depends on d, e, f
```

So, if `/var/s2` changes then the value of c should be updated. Updating c causes to recompute d and g.

The project is behind schedule. The colleague working on the parser and the operating system is sick. Shortcuts are in order. First, to avoid a parser, programs are in JSON. Second, you can simulate the file system.

Requirements

Implement a function `main()` that takes 3 arguments: JSON representing a program, JSON representing commands, and a printer (to be discussed below). The goal is to simulate the execution of the commands.

Programs

A program is a sequence of statements encoded in JSON. An example is:

```
[{"kind":"Watch",
  "variable":{"kind":"Var","name":"wt"},
  "file":{"kind":"Target","name":"/var/t"}},
 {"kind":"Watch",
  "variable":{"kind":"Var","name":"ws"},
  "file":{"kind":"Target","name":"/var/s"}},
 {"kind":"Simple",
  "variable":{"kind":"Var","name":"x"},
  "expression":{"kind":"Add",
                "left":{"kind":"Var","name":"wt"},
                "right":{"kind":"Var","name":"ws"}}}]
```

This encodes

```
wt = watch("/var/t")
ws = watch("/var/s")
x = wt + ws
```

There are three kinds of statements:

- simple assignments (`kind:"Simple"`),
- conditional assignments (`kind: "Conditional"`), and
- watched file assignments (`kind:"Watch"`).

Each of these has a target variable (`variable`). A simple assignment computes the value of an expression (`expression`) and assigns it to the variable. A conditional assignment checks if a guard variable is zero (`guard`), and assigns the value of either the first (t`rueexp`) or second expression (`falseexp`) to the variable. A `watch` statement represents a variable that is associated with a file. When the file is updated, the watch statement updates the value of the variable.

Expressions come in several kinds. Binary operations, `Add, Div, Mult` and `Sub`, have `left` and `right` subexpressions. Variables (`Var`) have a `name`. Numbers (`Num`) have a `value`. The last expression is a target (`Target`) with a `name` representing the file being watched.

Commands

Commands represent external interaction with your application, they are used to simulate the file system as it updates. There are two commands. The `Update` command specifies the name of a file and its updated value. This mimics a new sensor reading. The `Monitor` command installs a monitor on a variable. After the monitor is installed, every change to that variable is sent to the printer for debugging and testing.

Here are commands to monitor variable `x` and to update to files `/var/s` and `/var/t`.

```
[{"kind":"Monitor","name":"x"},
 {"kind":"Update","name":"/var/s","value":2},
 {"kind":"Update","name":"/var/t","value":3}]
```

Printer

The monitor command logs each time a variable is updated by calling method `log` of `IPrinter`:

```
interface IPrinter {
   log( msg: {var: string, val:number} ) : void
   last( name: string) : number
}
```

where `msg` has a field `var` containing a string (the name of the variable being updated) and `val` (it's value). Method `last()` returns the value of the last update to a variable or undefined if it was not updated.

Files

A file has a kind and a name, each of which is a string. The `kind` field is always `"Target"`.

```
{"kind":"Target","name":"/var/t"}
```

Plan of Attack

This problem requires you to do several things. Here is a possible plan of attack:

1. Represent statements and expressions as objects with classes like `Var` (an expression that is just a variable), `Num`, and classes to represent arithmetic. A hierarchy can be handy for 3 kinds of statements and 6 kinds of expressions (some are binary, some are unary).
2. Evaluate statements, this could be done by a method on each object or by a visit.
3. Write a function to take JSON and build a representation of the program it denotes.
4. Write another function to execute a sequence of commands.
5. Last, the problem requires **incremental** updates. When a variable's value changes, only those portions of the program that depend on it should be recomputed. For each variable, you will need to keep track of statements that depend on it. Computing these dependencies could be done by a Visitor which returns a set of variables that a statement depends on.
6. To implement incremental evaluation, we recommend to notify statements of change with an Observer. One possible approach is to have a class Binding that keeps the mapping between variables and their values, and have that class play the role of the Subject in the Observer pattern. You may use the same approach for files.

Excluding tests and comments, a compact solution is about 300 lines. Your solution can be any length, we mention this as a guide. If you find yourself writing much more code, talk about your solution in office hours. This problem gives you multiple opportunities to use the design patterns we will discuss in class. The assignment is being released **early** to allow you to ask questions in class. We recommend you start working on it before you finish reading about patterns and refactor your code as you learn how to fit patterns in it.

Deliverables

Implement a single file named `index.ts` that exports a `main` function and an `IPrinter` interface. You are **not** required to submit unit tests for this problem, but make sure that the provided tests pass. We will test your code on additional tests, and you are unlikely to pass them without thoroughly testing your code.

Question 1 (40 points) Correctness

Your implementation will be tested for correctness and incremental updates. We may review code for quality.

Question 2 (40 points) Use of Patterns

You are required to use the Visitor and Observer patterns, and two design patterns of your choice. For each pattern, provide an extract of your code that implements the pattern (make sure to include comments) and answers to with the following questions (10 pts/pattern):

- How does your code implement the pattern?
- What benefits can one expect from the pattern?

Question 3 (20 points) Short Answers

Briefly answer (5 pts/question):

1. When is it beneficial to use a Singleton?
2. How does the Observer pattern decrease coupling between classes?
3. What kinds of changes to the code are easier to make when using a Visitor?
4. What is the difference between a Builder and a Factory?

Answers to questions 2 and 3 should reside in a file named “Questions.PDF”

Testing

The following can be used as a starting point for testing.

```
import { main, IPrinter } from './index'
import * as fs from 'fs'
import { expect } from 'chai';
class Printer implements IPrinter {
  private lastValue: Map<string,number> = new Map<string,number>()
  log(msg: { var: string; val: number; }): void {
    this.lastValue.set(msg.var, msg.val)
  }
  last(name: string): number { return this.lastValue.get(name) }
}
const program1: any[] =
  JSON.parse(fs.readFileSync('./src/input/program1.json', { encoding: 'utf8' }))
const program2 : any[] =
  JSON.parse(fs.readFileSync('./src/input/program2.json', { encoding: 'utf8' }))
const commands1 : any[] =
  JSON.parse(fs.readFileSync('./src/input/commands1.json', { encoding: 'utf8' }))
const commands2 : any[] =
  JSON.parse(fs.readFileSync('./src/input/commands2.json', { encoding: 'utf8' }))
const commands3 : any[] =
  JSON.parse(fs.readFileSync('./src/input/commands3.json', { encoding: 'utf8' }))

describe('HW2', (): void => {
  it('should handle the example program from the assignment', (): void => {
    const printer = new Printer();
    main(program1, commands1, printer);
```

```
    expect(printer.last('x')).to.equal(5);
  });
  it('handle the example program from assignment with added commands', (): void => {
    const printer = new Printer();
    main(program1, commands2, printer);
    expect(printer.last('x')).to.equal(5);
  });
  it('handle the larger program from the assignment with some commands', (): void => {
    const printer = new Printer();
    main(program2, commands3, printer);
    expect(printer.last('a')).to.equal(7)
    expect(printer.last('b')).to.equal(14)
    expect(printer.last('e')).to.equal(7)
    expect(printer.last('g')).to.equal(1)
  })
});
```

input/commands1.json

```
[{"kind":"Monitor","name":"x"},
 {"kind":"Update","name":"/var/s","value":2},
 {"kind":"Update","name":"/var/t","value":3}]
```

input/commands2.json

```
[{"kind":"Monitor","name":"x"},
 {"kind":"Monitor","name":"ws"},
 {"kind":"Monitor","name":"wt"},
 {"kind":"Update","name":"/var/s","value":2},
 {"kind":"Update","name":"/var/t","value":3}]
```

input/commands3.json

```
[{"kind":"Monitor","name":"a"},
 {"kind":"Monitor","name":"b"},
 {"kind":"Monitor","name":"c"},
 {"kind":"Monitor","name":"d"},
 {"kind":"Monitor","name":"e"},
 {"kind":"Monitor","name":"f"},
 {"kind":"Monitor","name":"g"},
 {"kind":"Update","name":"/var/s1","value":7}]
```

input/program1.json

```
[{"kind":"Watch",
  "variable":{"kind":"Var","name":"wt"},
  "file":{"kind":"Target","name":"/var/t"}},
 {"kind":"Watch",
  "variable":{"kind":"Var","name":"ws"},
  "file":{"kind":"Target","name":"/var/s"}},
 {"kind":"Simple",
  "variable":{"kind":"Var","name":"x"},
  "expression":{"kind":"Add",
                "left":{"kind":"Var","name":"wt"},
```

```
                "right":{"kind":"Var","name":"ws"}}}]
```

input/program2.json

```
[{"kind":"Watch",
  "variable":{"kind":"Var","name":"a"},
  "file":{"kind":"Target","name":"/var/s1"}},
 {"kind":"Simple",
  "variable":{"kind":"Var","name":"b"},
  "expression":{"kind":"Mul",
                "left":{"kind":"Var","name":"a"},
                "right":{"kind":"Num","value":2}}},
  {"kind":"Watch",
    "variable":{"kind":"Var","name":"c"},
    "file":{"kind":"Target","name":"/var/s2"}},
  {"kind":"Simple",
    "variable":{"kind":"Var","name":"d"},
    "expression":{"kind":"Add",
                  "left":{"kind":"Var","name":"c"},
                  "right":{"kind":"Num","value":1}}},
  {"kind":"Simple",
    "variable":{"kind":"Var","name":"e"},
    "expression":{"kind":"Sub",
                  "left":{"kind":"Var","name":"b"},
                  "right":{"kind":"Var","name":"a"}}},
  {"kind":"Watch",
    "variable":{"kind":"Var","name":"f"},
    "file":{"kind":"Target","name":"/var/s3"}},
  {"kind":"Cond",
    "variable":{"kind":"Var","name":"g"},
    "guard":{"kind":"Var","name":"f"},
    "trueExp":{"kind":"Div",
               "left":{"kind":"Var","name":"e"},
               "right":{"kind":"Var","name":"f"}},
    "falseExp":{"kind":"Var","name":"d"}}]
```