Introduction to State Machines

Disclaimer

This is a practical introduction to the concept of state machines to support your programming needs for this course. This is not intended to be a formal tutorial, rigorous, or complete.

Intuition

In ordinary life context often affects how we respond to stimuli. For example, at some point you might realize you feel tired. If you are in bed with your head on a pillow, your response might be to close your eyes and go to sleep. However, if you are driving a car or flying a plane, hopefully you would choose a different response to that same feeling. In the case where you're driving, you might choose to change your context at the soonest opportunity, perhaps you would find a place to stop and then in that context take a nap. In the case where you're flying a plane, where you might not have the option of pulling over for a rest, you might respond by drinking a cup of coffee.

For writing programs, a similar pattern often helps work through the organization of a solution to a complicated problem. A "state machine" is a section of the code that has multiple states (contexts) and performs an action for each input. Those actions may include transitioning to another state.

Example

Here is little bit of the code matching the sleepiness example. (See below for an explanation of the switch-case statements used in this code.)

```
while(input) {
```

```
    switch(state) {
    case BED:
        switch(input) {
        case SLEEPY:
            close_eyes()
            state = SLEEPING;
            break;
        case HUNGRY:
            state = KITCHEN;
            eat();
            break;
        }
        break;
    case FLYING:
        switch(input) {
        case SLEEPY:
            drink_coffee(); // Failure of this operation may result in a crash
            break;
        case HUNGRY:
            eat();
            break;
        }
        break;
    }
}
```

Going back to the sentiment analysis task you implemented in an earlier assignment, you needed to process the score and tokens of a line from a file. Part of a state machine for such a task might is presented below. (See below for an explanation of the enum statement.)

```
enum STATES { START, SIGN, SCORE, TEXT, WORD, NON_WORD }

state = STATES.START;
score = 1;
for(char c: line.toCharArray()) {
    switch(state) {
    case START:
        switch(c) {
        case '-':
```

```
            score = -1;
        case '+':
            state = STATES.SIGN;
            break;
        case '0':
        case '1':
        case '2':
            score = c - '0';
            state = STATES.SCORE;
            break;
        default:
            fail();
        }
        break;
    case SIGN:
        switch(c) {
        case '0':
        case '1':
        case '2':
            score *= c - '0';
            state = STATES.SCORE;
            break;
        default:
            fail();
        }
        break;
    case SCORE:
        if (Character.isWhiteSpace(c))
            state = STATES.TEXT;
            break;
        else
            fail();
    case TEXT:
        if (Character.isWhiteSpace(c))
            break;
        else if(Character.isLetter(c)) {
            cur_word.append(c);
            state = STATES.WORD;
        } else
            state = NON_WORD;
```

```
        break;
    case NON_WORD
        if (Character.isWhiteSpace(c))
            state = STATES.TEXT;
        break;
    case WORD
        if (Character.isWhiteSpace(c))
            state = STATES.TEXT;
        else
            cur_word.append(c);
        break;
    }
}
```

Note: this is still pseudocode, and not intended to be complete.

Switch Statement

Switch statements are a compact way of branching for multiple possible values of a variable. They are more compact than writing many if-else statements.

Instead of writing:

```
if(a == 0) {
    // code for 0
} else if(a == 1) {
    // code for 1
} else if(a == 2) {
    // code for 2
} else {
    // code for any other value of a
}
```

You can write:

```
switc(a) {
    case 0: // code for 0
```

```
        break;
    case 1: // code for 1
        break;
    case 2: // code for 2
        break;
    default: // code for any other value of a
}
```

It might not look simpler at first, but your perspective may shift with experience. Aside from the visual appeal, switch is usually considered[1] more efficient than a long sequence of if-then-else. The optional break statements also let you define control flows that are not easy to do with if-then-else. Also, when used with enumerations, code checkers (such as those run in the background in Eclipse and IntelliJ) may warn if you forget to cover all cases.

See: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

Enumerations

The `enum` statement may be used to compactly define a set of [usually] related constants. The `STATES` enumeration in the example could be replaced with a bunch of individually declared and defined constants:

```
static final int START = 0;
static final int SIGN = 1;
static final int SCORE = 2;
// multiple declaration form:
static final int TEXT = 3, WORD = 4, NON_WORD = 5;
```

Enumerations have the added benefit that they limit values to just those defined, whereas those constants can be used with `int score`, but there is nothing preventing us from accidentally assigning other values to `score`. In Java enumerations can act mostly like classes, the main difference being that all instances must be defined in the class declaration.

See: https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

[1] We can't rule out the possibility that the compiler or runtime optimizer will automatically convert the if-then-else chain into a switch for you.

A Note on the Topic of Regular Expressions

If you think you can avoid state machines by using regular expressions, think again. Regular expression are equivalent to a class of state machines called "non-deterministic finite automata" (NFA). A common approach to making a regular expression engine is to compile an expression into a state machine at runtime. Those implementations may not be as efficient as a custom state machine written specifically for your task.

Experience explicitly writing state machines can also be beneficial to thinking through writing regular expressions.