Lecture Note

Regular Expressions

State Machines

State Pattern

Finn E. Nordbjerg

Contents

1 Regular languages and regular expressions

Regular languages (or regular expressions) are a family of formal languages with a simple structure. They are among others useful for string matching, for instance input validation (e-mail addresses, dates, URLs etc.), for parsing protocols and much more.

As all formal languages, regular languages are defined over some finite alphabet (character set). All valid expressions in a regular language must be constructed using only concatenation, selection and iteration of symbols in the alphabet of the languages. This means that regular expressions cannot have nested parenthesis, nested blocks etc.

Examples of regular languages are valid integers, valid decimals and valid identifiers in a programming language like Java, or valid email addresses, valid URLs or valid IP-addresses. On the other hand, arithmetic expressions cannot be defined as regular expressions (nested parenthesis), either can programming languages like Java (nested '{---}' code blocks).

Most modern programming languages include libraries to handle regular expressions (for instance java.util.regex – see [1]).

Formally, a regular language over some alphabet can be defined as the set of strings of symbols from the alphabet that can be formed from the corresponding regular expression. A regular expression *r* over some alphabet is defined by:

1. $r = \varepsilon$ //*empty*
2. $r = a$ // *a* is a symbol from the alphabet
3. $r = st$ //concatenation of two regular expressions, *s* and *t*
4. $r = s \mid t$ //selection between two regular expressions *s* and *t* ("*s* or *t*")
5. $r = (s)^*$ //iteration: 0 or repetitions of the regular expression *s*

Note that the definition is recursive; rule 1 and 2 constituting the base case.

1.1 Example: Unsigned integers

If we want to define the unsigned integers using a regular expression, the alphabet could be:

[0-9] //one of the characters '0' to '9'

[0-9] is a shorthand for the regular expression 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Then unsigned integer may be defined as:

unsignedInteger:

[0-9]([0-9])*

In other words: An unsigned integer is a digit followed by zero or more digits.
We can enhance the regular expression so that signed integers are defined:

signedInteger:

(+ | -)[0-9]([0-9])*

If we still what to include unsigned integers, we may use the empty expression ε, and write:

signedInteger:

(ε | + | -) [0-9]([0-9])*

This regular expression defines integers as 123, +123 and -123.

(Is 0123 also allowed?)

1.3 Exercise: Decimals, scientific notation and identifiers

a) Write a regular expression that defines decimals (for instance, -123.456).
b) Write a regular expression that defines numbers in scientific notation (for instance, 123456.789 can be written as 1.23456789 E+5 using scientific notation).
c) Write a regular expression that defines valid Java identifiers (recall that an identifier is a string that can be used as a name, for instance for a class, a method or a variable)

1.4 Regular expressions in Java

For practical purposes this very simple notation is often enhanced, for instance *(r)+* is used meaning 1 or more repetitions of *r*, and *(r)?* is used meaning 0 or 1 occurrences of *r*.

When using regular expressions always check the notations used in the API, you are going to use in the case of Java check the documentation to java.util.regex ([2]).

For instance, in Java notation valid signed integers may be defined by this regular expression (pattern):

```
String numberPattern = "^(\\+|-)?[0-9]+$";
```

The enclosing '^' and '$' mean that there is not allowed anything else in the string to be validated. The backslash ('\') before the '+' is used to tell that we mean the symbol '+', not the meta symbol '+' (meaning 1 or more). So '\+' means the symbol '+' to java.util.regex. The second backslash is necessary in order to escape the in the Java string (so meta symbols are escaped to java.util.regex using '\' and the '\' is escaped to Java strings using the second '\'. In total a symbol that is also used as a meta symbol (like '+') must be prefixed with two backslashes in the Java string.

See demo: *RedExDemo*.

Java provides the java.util.regex package for pattern matching with regular expressions. The java.util.regex package primarily consists of the following three classes:

- ***Pattern Class:*** *A Pattern object is a compiled representation of a regular expression. To create a pattern, invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.*
- ***Matcher Class:*** *A Matcher object is the engine that interprets the pattern and performs match operations against an input string. You obtain a Matcher object by invoking the matcher method on a Pattern object.*
- ***PatternSyntaxException:*** *A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.*

(From [1] also see and [4]).

The APIs (including java.util.regex) also provides methods to extract all substrings in a string that matches a given regular expression (see the documentation).

1.5 Exercise: Using java.reg.ex

a) Create Java programs that check some of the regular expressions from exercise 1.2.
b) Define regular expressions to check valid URLs and email addresses
c) Write a Java program that can all extract URLs and/or email addresses from any text (a string).

2 State machines

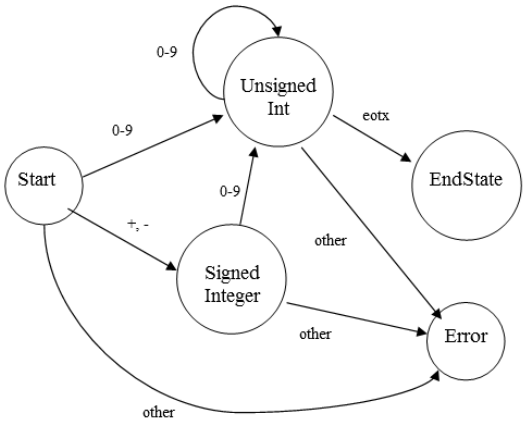
Another way of checking if a string matches a regular expression is to build a state machine that recognises the regular language.

A state machine is a set of input (e.g. symbols, events), a set of states and a transition function that takes a state and an input as argument and returns next state. (A state machine is sometimes called a DFA: Deterministic Finite Automaton).

Often, a state machine is given graphically as a state diagram (a digraph), where states are represented by nodes and transitions by edges. Transitions (edges) are labelled with input.

2.1 Example: A state machine that accepts signed integers:

For example, a state machine that recognises (accepts) signed integers:



Using this graph, a string of input symbols is checked using this algorithm:

```
currentState= Start;
currentInput= first symbol in the input string;
while not Eotxt and not State == Error do
        if there exists an edge from currentState marked with currentInput
        then    currentState= the state pointed to by that edge
                currentInput= next symbol in the input string
        else
                currentState= Error;
        endif;
endwhile;
if currentState = Error then error in input string else input string ok endif
```

So, the state machine is passed a string with input symbols and will determine, if that string is a valid signed integer or not.

It starts in the start state. From the current state and the current input symbol a transition to the next state is chosen. This continues until *Error* or *EndState* is reached.

2.2 State machines and regular expressions

There is a one-to-one correspondence between state machines and regular expressions: every string defined by a regular expression may be accepted by a state machine (and visa verse). Given a regular expression, it is possible to automatically construct a state machine that accepts strings defined by that regular expression (see [5]). This is actually what is used in the APIs (as java.util.regex).

2.3 Exercise: State machine that accepts decimal

Construct (state diagram) a state machine that accepts:

a) Decimal numbers
b) Scientific numbers
c) Identifiers

(cf. exercise 1.3).

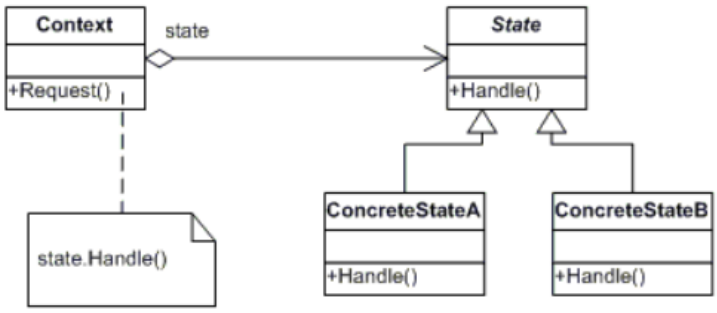
2.4 Implementing state machines: State Pattern

Next state is defined by as a function taken current state and current input as arguments and returning next state. Often the possibility of associating an action with the transition (or the state) is added to the state machine.

Different approaches to implementing this are possible. We shall focus on an object-oriented approached called *State Pattern*.

The core of the pattern is that a general state is an abstract class defining the transition function (and, if relevant, the action as well) as abstract methods. The actual states are classes that inherit from the abstract class and implement the abstract methods according to possible input, transitions from the state and actions associated with the state.

The design is shown by this UML class diagram (from [6]):

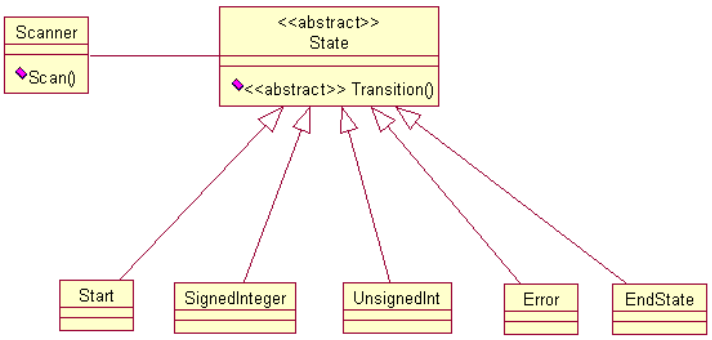


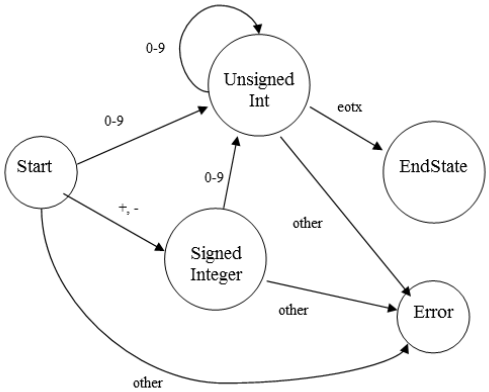
The interface to the state machine is the class *Context*. It holds the states of the state machine. The concrete states are all subtypes of the abstract class *State*. *State* defines the abstract methods for handling transitions and actions (if any), here represented by *Handle()*. The concrete states are represented by classes (*ConcreteStateA* and *ConcreteStateB*) that inherit from *State* and implement the abstract methods. The *Context* class holds a reference of type *State* (*currentState*) that references the actual state. Note how polymorphism allows *currentState* to reference all the subtypes of *State*. For deeper description of State Pattern, see [7].

2.5 Example: State pattern and signed integer

Now, let's use the pattern to implement the state machine that recognises signed integers (the context is called *Scanner* because software that recognises strings of different patterns in compilers is called a scanner).

In order to implement the integer scanner we will need the abstract class State and five concrete classes representing the states of the machine. Below we have a class diagram (UML) and the state diagram.





The code for *State* looks like this:

```
public abstract class State {

	//returns next state
	abstract public State transition(char c);

	//in case there are any actions connected to a transition
	//abstract public void action(char c);
}
```

Input symbols are simple chars.

The concrete states have references to the states to which they are directly connected, and they implement *transition()* (and, if any, *action()*).

For instance is *UnsignedInt* implemented like this:

```
public class UnsignedInt extends State {
	private State uSign, end, err;

	public void setTransitions(State uSign, State end, State err){
		this.uSign= uSign;
		this.end= end;
		this.err= err;
	}

	@Override
	public State transition(char c){
		if (c == '\0')
			return end;
		else if ('0' <= c && c <= '9')
			return uSign;
		else
			return err;
	}
}
```

The class *Scanner* holds the machine: It has references to the concrete states:

```
public class Scanner {
	private Start start;
	private UnsignedInt uSign;
	private SignedInteger sign;
	private Error err;
	private EndState end;
```

The constructor sets up the machine:

```
public Scanner(){
	start = new Start();
	uSign = new UnsignedInt();
	sign = new SignedInteger();
	err = new Error();
	end = new EndState();
	start.setTransitions(uSign, sign, err);
	uSign.setTransitions(uSign, end, err);
	sign.setTransitions(uSign, err);
}
```

The central scanner loop (section 2.1) is implemented by this method in class *Scanner*:

```
public boolean scan(String input){
	//input.length()>0
	boolean ok = false;
	int i = 0;
	State currState = start;
	while (currState != end && currState != err) {
		char nextChar;
		if (i == input.length())
			nextChar = '\0';
		else
			nextChar = input.charAt(i);
		//currState.Action();
		currState = currState.transition(nextChar);
		i++;
	}
	if (currState == end)
		ok = true;
	return ok;
}
```

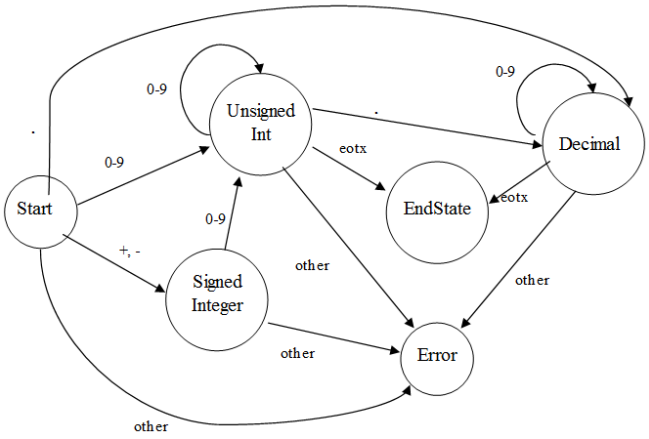
See the demo: *IntegerScanner*.

2.6 Critique of the design

This implementation works fine, if the states and transitions are rather static (as in recognising regular expression), but if new states and transitions may be added during the lifetime of the program, the implementation is vulnerable. A lot of small pieces of code must be added and change in many different classes. This makes maintenance difficult. Try to do the following exercise and see how many additions and changes you have to do:

2.7 Exercise: State pattern and decimals

Implement a state machine that recognises decimal numbers. Use your solution to exercise a) in section 2.3 or this diagram:



Make a copy of the demo (*IntegerScanner*) from section 2.5, and modify that in your solution. Note how many different places in the existing code, you have to add or change code.

2.8 Alternative design

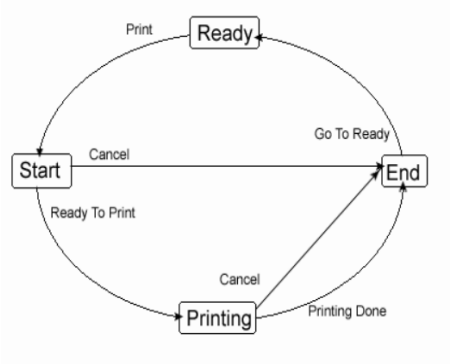
A better design would be, if state and transition handling could be isolated, so adding new states and transition wouldn't require changes in the existing concrete state classes. If states and transition are handled by the abstract class *State* and inherited by the concrete states, then we will still need to add the new states, but adding and changing transitions could be isolated to the context class.

We will illustrate this by an example (which, by the way, has nothing to do with scanning). This example illustrates another area of usage of state machines: Regulation and control.

2.9 Example: Controlling a printer

The following example is based on [8].

A simplified control for printer could be done by this state machine:



Transitions are mark with events that may occur during the operation of the printer, either as user input or as events from the printer itself.

If the state pattern is used to implement this state machine, we will get the following classes:

- *State* (abstract)
- *Start*
- *Ready*
- *Printing*
- *End*
- *PrinterController* (the context class)

In order to simulate the actual printer, events are simply defined as this enum in the context class (*PrinterController*):

```
enum Events { print, goReady, done, cancel, readyToPrint, unknown };
```

The handling of transitions is implemented as a *Map* in the abstract class *State*:

```
public abstract class State {

	private Map<Events, State> adjacents;

	public State() {
		adjacents = new TreeMap<Events, State>();
	}

	public Map<Events, State> getAdjacents() {
		return adjacents;
	}

	public void addTransition(Events e, State s) {
		adjacents.put(e, s);
	}
```

This map is inherited by every concrete state, so in this way each concrete state has a map that associates an event with the next state corresponding to this event.

The transition function may be implemented in *State* leaving only *action()* abstract:

```
	//get next state
	public State transition(Events e) {
		return getAdjacents().get(e);
	}
	//Actions associated with the concrete state
	public abstract void action();
}
```

The machine is (as before) constructed by the constructor of the context class (*PrintController*). The context class also (as before) defines the concrete states:

```
class PrintController {
	private Start  start;
	private Printing printing;
	private Ready ready;
	private End end;

	public PrintController() {
		start = new Start();
		printing = new Printing();
		ready = new Ready();
		end = new End();
```

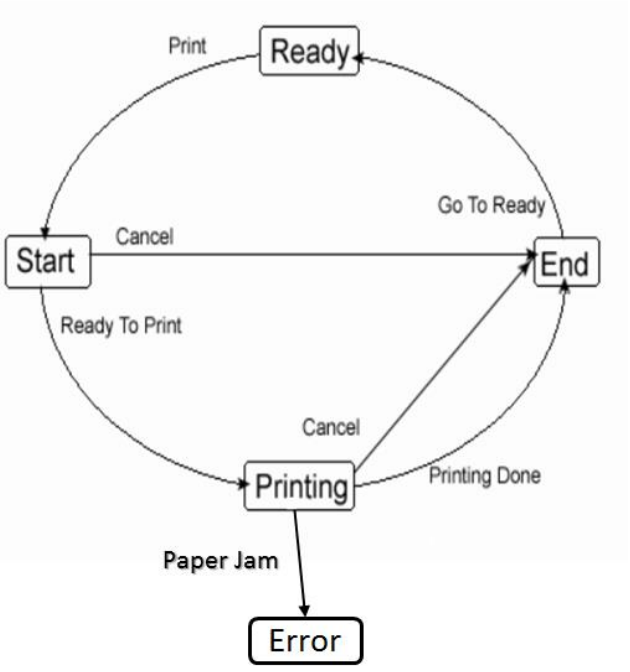
```
        start.addTransition(Events.readyToPrint, printing);
        start.addTransition(Events.cancel, end);

        printing.addTransition(Events.cancel, end);
        printing.addTransition(Events.done, end);

        ready.addTransition(Events.print, start);

        end.addTransition(Events.goReady, ready);
    }
```

The rest of the design remains the same. See the demo: PrinterStates.

2.10 Exercise: Extending the printer controller

Add a new event that may occur during printing: “Paper Jam”. The event can occur in the state “Printing” and brings the printer in an error state where it remains stopped.

The new state machine could be described by this diagram:



Make a copy of the demo *PrinterStates* and modify the code in order to implement the machine above.

Note how many places you have to add or change code. Compare to section 2.7 and reflect☺.

3 Code samples (demos)

1. RegExDemo
2. IntegerScanner
3. PrinterStates

4 References

[1]: http://www.tutorialspoint.com/java/java_regular_expressions.htm (accessed 26-02-2016).

[2]: https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html (accessed 24-02-2016)

[3]: http://www.tutorialspoint.com/java/java_regular_expressions.htm (accessed 26-02-2016).

[4]: http://www.sw-engineering-candies.com/blog-1/howtofindvalidemailaddresswitharegularexpressionregexinjava) (accessed 26-02-2016).

[5]: Kenneth C. Louden: *Compiler Construction. Principles and Practice*. PWS Publishing Company 1997. ISBN 0-534-93972-4.

[6]: http://www.dofactory.com/net/state-design-pattern (accessed 26-02-2016).

[7]: Mark Grand: *Patterns in Java, Vol. 1.* Wiley 1998. ISBN 0-471-25839-3

[8]: http://www.go4expert.com/forums/showthread.php?t=5127 (accessed 27-02-2016).