Regular Expressions in Smalltalk

Just Like Haskell

One subclass for each alternative representation

Write Tests

1. Run tests
2. Get message not understood
3. Define method
4. Repeat from 1
... 19. Get real failure
Write Tests

What's the problem?

I need an instance, not the class

- But there need be only one instance of \texttt{REEmpty}
- Enter: the Singleton pattern.
  - make a class instance-variable called \texttt{uniqueInstance}
  - make a class-side method named \texttt{default}
    
    ```
    Default
    
    uniqueInstance = nil
    uniqueInstance
    ```
    - override \texttt{new} to be an error

What do we have so far?

Convenience Operations

- Write tests:
  - \texttt{self assert: $a \text{asRE printString} = 'a'}
  - \texttt{self assert: (a + b) printString = 'a+b'}
- Why compare \texttt{printStrings}?
Where do the operation methods go?

- In the abstract superclass `RegularExpression`
  - so that they work for all the subclasses

Refactor tests to remove duplication

```smalltalk
assert: anExpression printsAs: aPrinting
self assert: anExpression printString = aPrinting
```

meaning1: sets of strings

- Code very similar to Tim’s Haskell version
- Only tricky part is star
  - Haskell version:

```haskell
meaning1 (Star r) = norm (zero ++ one ++ two ++ three)
where zero = ['"]
    one = meaning1 r
    two = [x++y | x ← one, y ← one]
    three = [x++y | x ← one, y ← two]
```

Smalltalk

```smalltalk
meaning1 [ zero | one | two | three | four | five | six | seven | eight | nine | ten | eleven | twelve | thirteen | fourteen | fifteen ]
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
```

which brings us to…

```
meaning1 = sets of strings
```

Smalltalk

```smalltalk
meaning1 [ zero | one | two | three | four | five | six | seven | eight | nine | ten | eleven | twelve | thirteen | fourteen | fifteen ]
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
result = String
```
Cross tests

- introspect on the instance variables of the test case
- select those that respond to the meaning1 message
- check that for every string str in re meaning1
  - re meaning2: str is true

Now RE’s pass the tests

Finite State Machines

**FINITE AUTOMATA AND REGULAR GRAMMARS**

### 3.1 THE FINITE AUTOMATION

In Chapter 2, we were introduced to a generating scheme—the grammar. Grammars are finite specifications for languages. In this chapter we shall see another method of fairly specifying input languages—the recognizer. We shall consider what is essentially the reverse recognizer, called a finite automaton. The finite automata (FAs) currently defined all languages defined by grammars, but we shall show that the languages defined are exactly the type 3 languages. In later chapters, the reader will be introduced to recognizer for type 2, 1, and 0 languages. Here we shall define a finite automaton as a formal system, that give the physical meaning of the definition.

A finite automaton $F$ over an alphabet $Σ$ is a system $(Q, Σ, δ, q_0, F)$, where $Q$ is a finite, nonempty set of states, $Σ$ is a finite input alphabet, $δ$ is a mapping of $Q \times Σ$ into $Q$, $q_0$ is the initial state, and $F \subseteq Q$ is the set of final states.

Our model (Fig. 3.1) represents a finite control which reads symbols from a linear input tape in a sequential manner from left to right. The set of states $Q$ consists of the states of the finite control. Initially, the finite control is in state $q_0$, and is scanning the leftmost symbol of a string of symbols in $Σ$ which enters on the input tape. The interpretation of $δ(a, σ) = q$ for a