Hamming’s problem

• h is an “Ordered Set”
• $1 \in h$
• $x \in h \Rightarrow 2^x \in h, 3^x \in h, 5^x \in h.$
• generate all elements of $h < \text{limit}$
Let’s solve it

1. Write a test
2. make the test run green
3. clean up the code
   ➡ remove any duplication
4. repeat until done
CS410/510 Advanced Programming
Lecture 7:

Regular Expressions in Smalltalk
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "++++showU x++++"++++showU y++++"
      where showU (Union x y) = show x++++"++++showU y
      showU x = show x
  show (Concat x y) = show x++++show y
  show (Star (x@((Concat _ _)))) = "++++show x++++"*"
  show (Star (x@((Union _ _)))) = "++++show x++++"*"
  show (Star x) = show x++++"*"
data RE
   = Empty
   | Union RE RE
   | Concat RE RE
   | Star RE
   | C Char

instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "("++showU x++"++"++showU y++")"
  where showU (Union x y) = show x++"++show y
  show (Concat x y) = show x++show y
  show (Star x) = "("++show x++")"
  show (Star (x@(Concat _ _))) = "("++show x++")"

```
setLeft: RE1 right: RE2
left := RE1.
right := RE2.
+ self
```

```
RB: REConcat

Connectors: Basic Distro
GS10ap Regular Express
Dandelion-util
Dandelion-exceptions
Dandelion-event
Dandelion-introspector
Dandelion-analysis-store
Dandelion-introspect-str
Dandelion-analysis-strat
Dandelion-resolve-strat
Dandelion-observable
Dandelion-observable
instance
?
class

impl vers inher hier iVar cVar source

-- all --
private
printing

printOn:
setLeft: right:

senders

browse

4
data RE
  = Empty
  | Union RE RE
  | Concat RE RE
  | Star RE
  | C Char

instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "(" ++ showU x ++ "+" ++ showU y ++ "")"
  where showU (Union x y) = show x ++ "+" ++ show y
  show (Concat x y) = show x ++ show y
  show (Star (x@((Concat _ _)))) = "(" ++ show (x@((Union _ _))) ++ "+" ++ show x
  show (Star x) = show x ++ "*"
Write Tests
Write Tests

1. Run tests
2. get message not understood
3. define method
4. repeat from 1
   ...
19. get real failure
Write Tests
Write Tests
What’s the problem?
I need an instance, not the class

• But there need be only one instance of REEmpty

• Enter: the Singleton pattern.
  • make a class instance-variable called uniqueInstance
  • make a class-side method named default
    ```
    default
    uniqueInstance ifNil: [uniqueInstance := self basicNew]. + uniqueInstance
    ```
  • override new to be an error
What do we have so far?
Convenience Operations

• Write tests:
  
  self assert: $a$ asRE printString = 'a'
  
  self assert: (a + b) printString = 'a+b'
  
• Why compare printStrings?
Where do the operation methods go?

- In the abstract superclass `RegularExpression`
  - so that they work for all the subclasses
Where do the operation methods go?

• In the abstract superclass RegularExpression
Refactor tests to remove duplication

**testPrinting**

```
self assert: epsilon printsAs: '#'.
self assert: a printsAs: 'a'.
self assert: b printsAs: 'b'.
self assert: aORb printsAs: 'a+b'.
self assert: ab printsAs: 'ab'.
self assert: abStar printsAs: 'ab*'.
```

**assert: anExpression printsAs: aprintString**

```
self assert: anExpression printString = aprintString
```
which brings us to…
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

REStar

```
meaning1
| zero one two three |
zero := '',
one := base meaning1.
two := self anyOf: one followedByAnyOf: one.
three := self anyOf: one followedByAnyOf: two.
↑ (Set with: zero) addAll: one;
    addAll: two;
    addAll: three;
yourself
```

RegularExpression

```
anyOf: m1 followedByAnyOf: mr
| result |
result := Set new.
m1 do: [:i | mr do: [:r | result add: i , r]].
↑result
```
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 AUTOMATA
AND REGULAR GRAMMARS

3.1 THE FINITE AUTOMATON

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 finitely specifying infinite languages—the recognizer. We shall consider what is undoubtedly the simplest recognizer, called a finite automaton. The finite automaton (fa) cannot define 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 recognizers for type 0, 1, and 2 languages. Here we shall define a finite automaton as a formal system, then give the physical meaning of the definition.

A finite automaton \(M\) over an alphabet \(\Sigma\) is a system \((K, \Sigma, \delta, q_0, F)\), where \(K\) is a finite, nonempty set of states, \(\Sigma\) is a finite input alphabet, \(\delta\) is a mapping of \(K \times \Sigma\) into \(K\), \(q_0\) in \(K\) is the initial state, and \(F \subseteq K\) is the set of final states.

Our model in 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 \(K\) 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 \(\Sigma\) which appear on the input tape. The interpretation of \(\delta(a, a) = n\) for \(a\)
The code with NFSM