

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

Just Like Haskell

```
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++"*"
```

Just Like Haskell

```
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++"*"
```

The screenshot shows a Haskell IDE window titled "RB: REConcat". The window is divided into several panes. The top-left pane shows a list of modules, with "CS510ap-RegularExpressi" selected. The top-right pane shows a list of functions, with "setLeft:right:" selected. The bottom pane shows the definition of the "setLeft:right:" function:

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

Below the main pane, there are several buttons: "browse", "senders", "impl", "vers", "inher", "hier", "iVar", "cVar", and "source".

Just Like Haskell

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

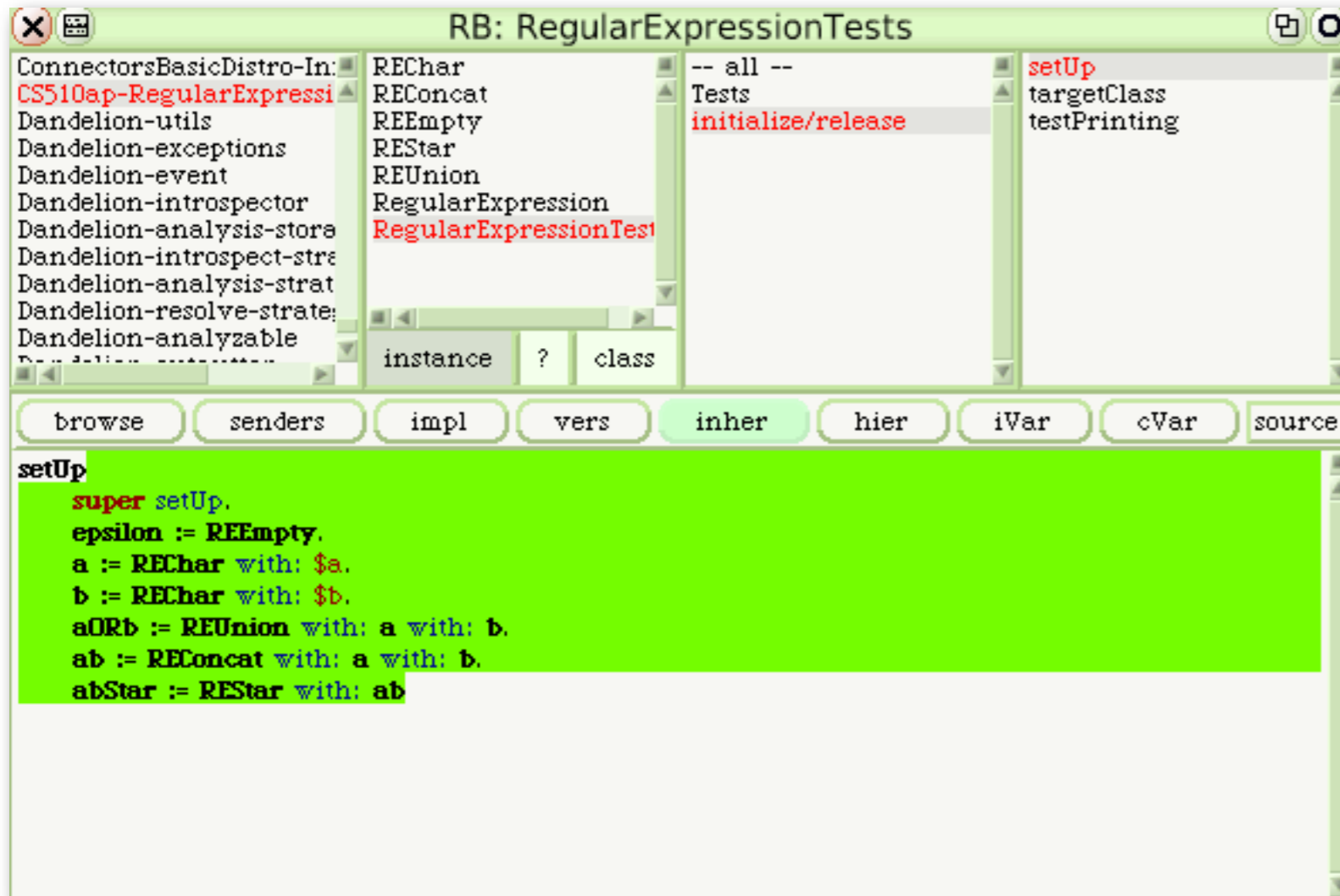
```
instance Show RE where
  show Empty = "#"
  show (C x) = [x]
  show (Union x y) = "("++showU x++"+"++showU y++")"
    where showU (Union x y) = show x++"+"++showU y
          showU x = show x
  show (Concat x y) = show x++show y
  show (Star (x@(Concat _ _))) = "("++show x++"*"
  show (Star (x@(Union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

One subclass for each alternative representation

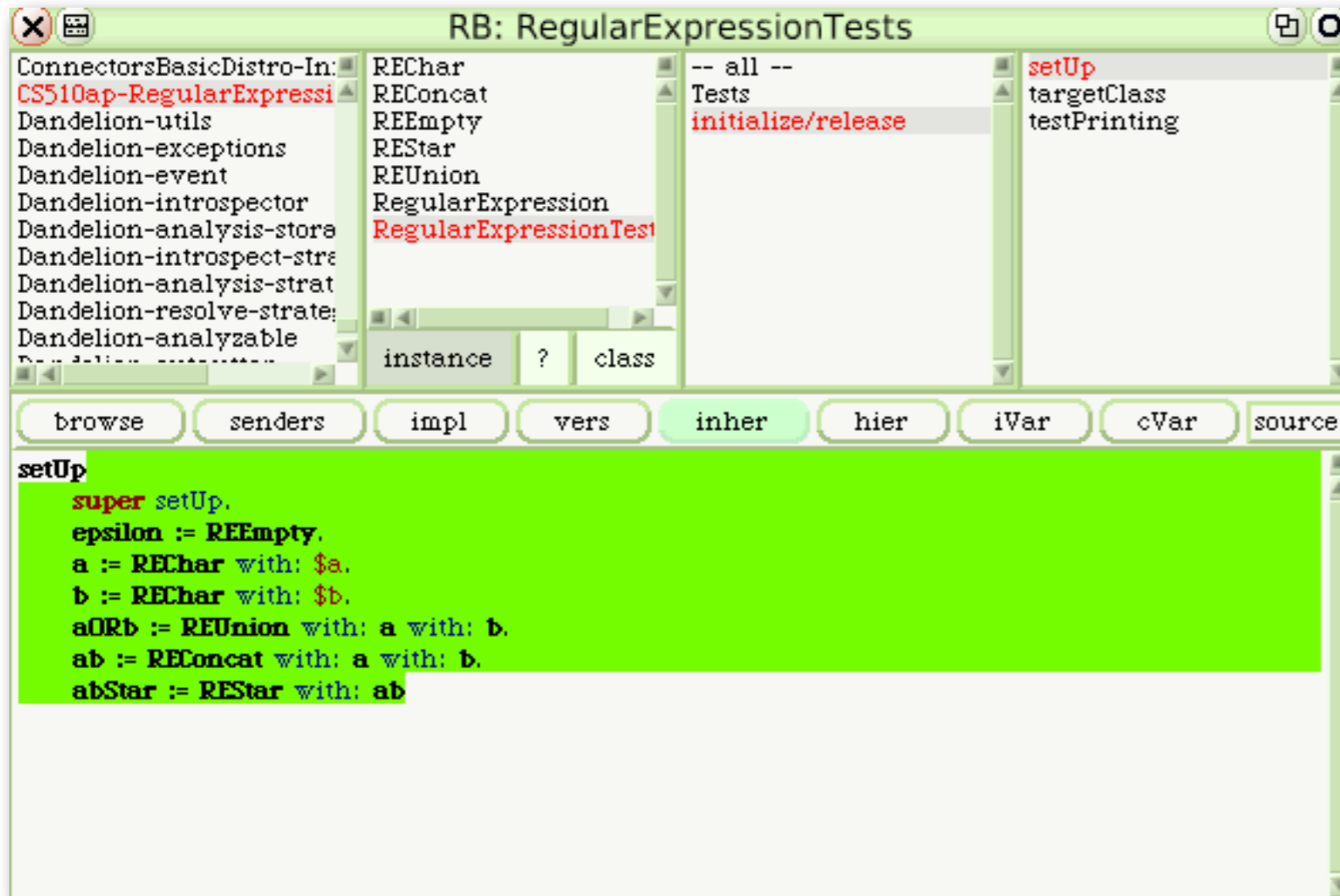
The screenshot shows the Haskell IDE (Hugs) window titled "RB: REConcat". The main pane displays a list of classes and instances, with "REConcat" selected. The list includes "REChar", "REConcat", "REEmpty", "REStar", "REUnion", "RegularExpression", and "RegularExpressionTest". The "REConcat" class is highlighted in red. Below the list, there are buttons for "instance", "?", and "class". The bottom pane shows the definition of the "setLeft" method for the "REConcat" class:

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

Write Tests

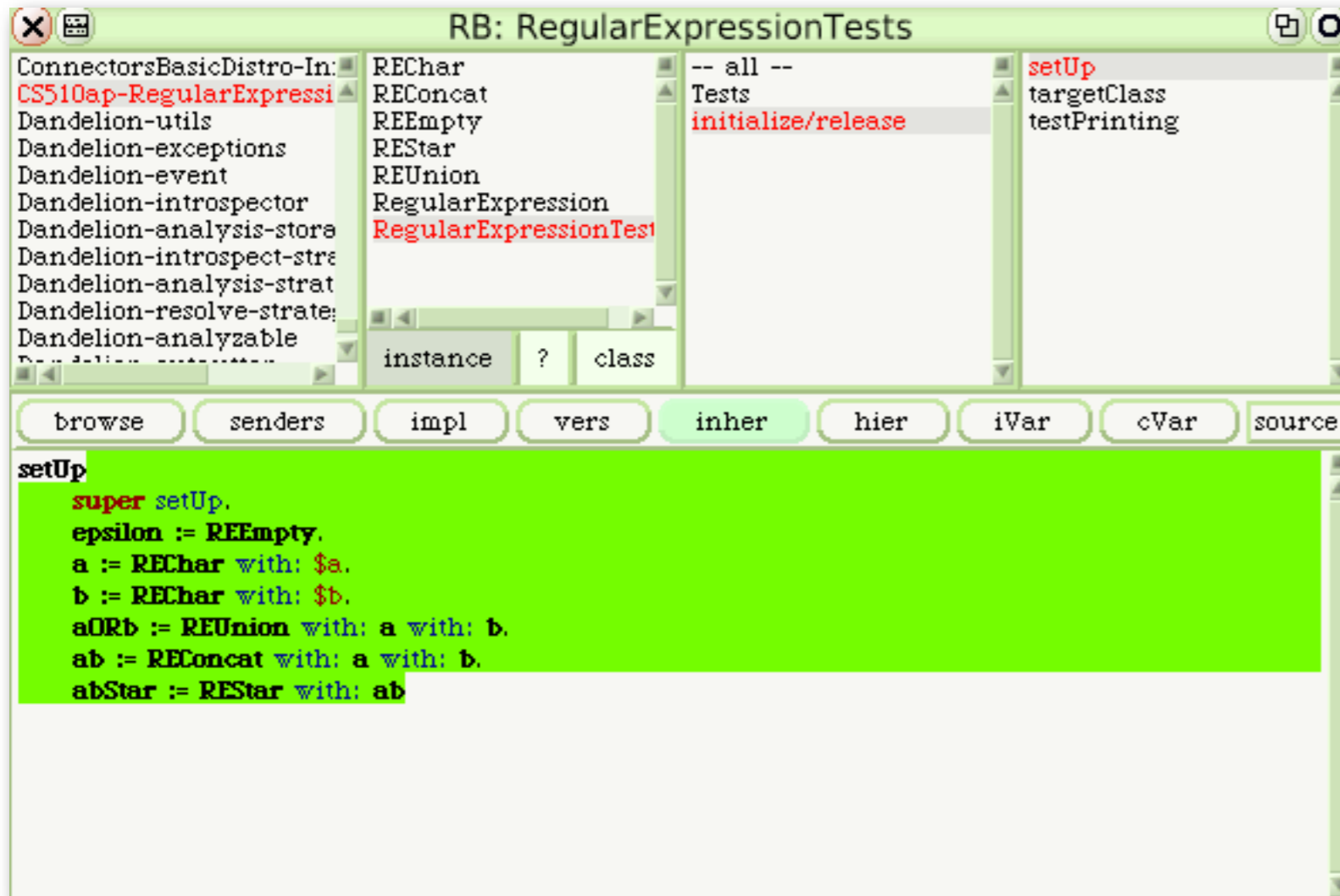


Write Tests

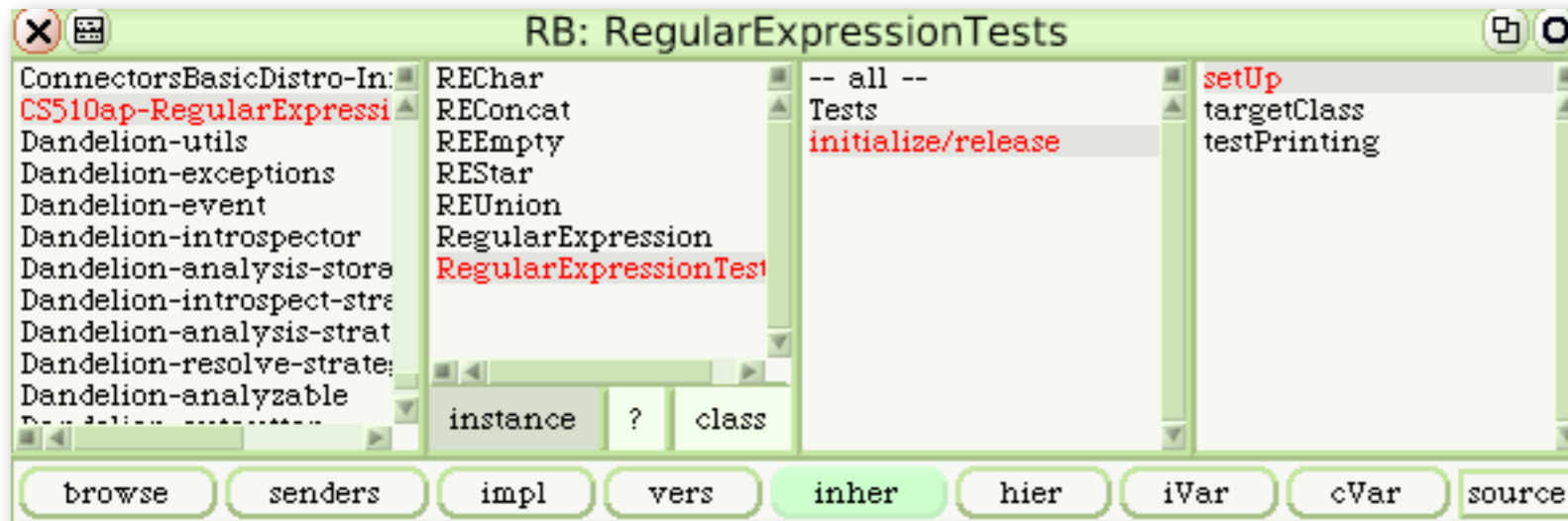


1. Run tests
2. get *message not understood*
3. define method
4. repeat from 1
- ...
19. get real failure

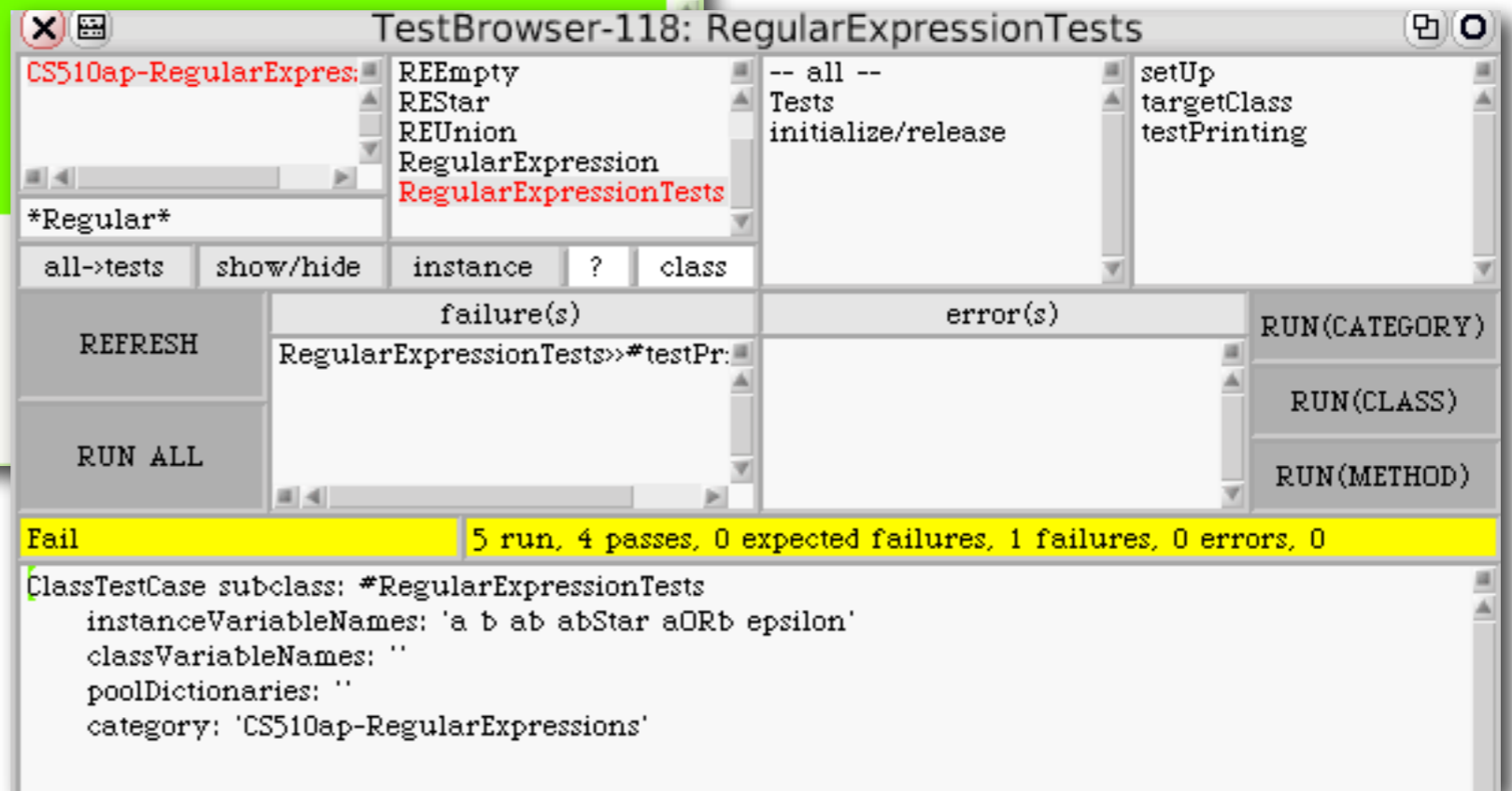
Write Tests



Write Tests



```
setUp
super setUp.
epsilon := REEmpty.
a := REChar with: $a.
b := REChar with: $b.
aORb := REUnion with: a with: b.
ab := REConcat with: a with: b.
abStar := REStar with: ab
```



TestFailure: Assertion failed

```

RegularExpressionTests(TestCase)>>signalFailure:
RegularExpressionTests(TestCase)>>assert:
RegularExpressionTests>>testPrinting
RegularExpressionTests(TestCase)>>performTest
[] in RegularExpressionTests(TestCase)>>runCase {[self setUp. self performTest]}
BlockContext>>ensure:
BlockContext>>sunitEnsure:
RegularExpressionTests(TestCase)>>runCase
[] in RegularExpressionTests(TestCase)>>debug {[self class selector: testSelector) runCase]}

```

apb 4/21/2006 12:02 · initialize/release · 5 implementors · only in change set RegEx ·

testPrinting

```

self assert: epsilon printString = '#'.
self assert: a printString = 'a'.
self assert: b printString = 'b'.
self assert: aORb printString = 'a+b'.
self assert: ab printString = 'ab'.
self assert: abStar printString = 'ab*'.

```

self
all inst vars
testSelector
a
b
ab
abStar
aORb
epsilon

REEmpty

thisContext
all temp vars

setUp

```

super setUp.
epsilon := REEmpty.
a := REChar with: $a.
b := REChar with: $b.
aORb := REUnion with: a with: b.
ab := REConcat with: a with: b.
abStar := REStar with: ab

```

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?

Dandelion
[Overview](#)
[Whole Index](#)

[All Classes](#)

All Categories

All Classes

- [REChar](#)
- [REConcat](#)
- [REEmpty](#)
- [REStar](#)
- [REUnion](#)
- [RegularExpresio](#)
- [RegularExpresio](#)

[^top](#)

- made by Dandelion

=

Dandelion

All Categories

[CS510ap-RegularExpressions](#)

All Globals

[ActiveEvent](#)

[ActiveHand](#)

[CustomEventsRegistry](#)

[Display](#)

[ImageImports](#)

[Processor](#)

[ScheduledControllers](#)

[ScriptingSystem](#)

[Sensor](#)

[Smalltalk](#)

[SourceFiles](#)

[SystemOrganization](#)

[TestConstants](#)

[Transcript](#)

Convenience Operations

```
alpha = Union (C 'a')
          (Union (C 'b') (C 'c'))
digit = Union (C '0')
          (Union (C '1') (C '2'))
key = Union (string "if")
          (Union (string "then")
                (string "else"))
punc = (C ',')
ident = Concat alpha
          (Star (Union alpha digit))
number = Concat digit (Star digit)
lexer = Union ident (Union number (Union key punc))

val re1 = Concat(Union (C '+')(Union (C '-')Empty))
          (Concat (C 'D')(Star (C 'D'))))

string :: String -> RE
string [] = Empty
string [c] = C c
string (c:cs) = Concat (C c) (string cs)
```

- 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`

The screenshot shows a TestBrowser window titled "TestBrowser-118: RegularExpression". The window is divided into several sections:

- Class Hierarchy:** A tree view on the left shows the class hierarchy: `REEmpty`, `REStar`, `REUnion`, `RegularExpression` (highlighted), and `RegularExpressionTests`.
- Search:** A search bar contains the text `*Regular*`.
- Buttons:** Below the search bar are buttons for `all->tests`, `show/hide`, `instance`, `?`, and `class`.
- Test Results:** A table with columns `failure(s)` and `error(s)`. The table is currently empty.
- Summary:** A green bar at the bottom of the table area displays the text: `Pass 10 run, 10 passes, 0 expected failures, 0 failures, 0 errors, 0`.
- Test Output:** A text area at the bottom shows the output for the `star` test: `star` and `↑ REStar with: self.`

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...

The screenshot displays the Dandelion web application interface. On the left is a sidebar with navigation links: [DandelionOverview](#), [Whole Index](#), [All Classes](#), and [All Categories](#). Below these is a section titled "All Classes" containing a bulleted list of class names: [REChar](#), [REConcat](#), [REEmpty](#), [REStar](#), [REUnion](#), [RegularExpresio](#), and [RegularExpresio](#). At the bottom of the sidebar are links for [^top](#) and [- made by Dandelion -](#).

The main content area is titled "Dandelion" and features two sections: "All Categories" and "All Globals". The "All Categories" section contains a single entry: [CS510ap-RegularExpressions](#). The "All Globals" section contains a list of global objects: [ActiveEvent](#), [ActiveHand](#), [CustomEventsRegistry](#), [Display](#), [ImageImports](#), [Processor](#), [ScheduledControllers](#), [ScriptingSystem](#), [Sensor](#), [Smalltalk](#), [SourceFiles](#), and [SystemOrganization](#).

meaning1: sets of strings

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

```
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
```

- Complicated enough to need a helper method

RegularExpression

```
anyOf: ml followedByAnyOf: mr
| result |
result := Set new.
ml do: [:l | mr do: [:r | result add: l , r]].
↑result
```

- Is there a simpler way to calculate * ?

Cross tests

```
Pass 17 run, 17 passes, 0 expected failures, 0 failures, 0 errors
```

```
testMeaning1AgainstMeaning2
```

```
self instanceVariableValues select: [ :each | each respondsTo: #meaning1 ] thenDo:  
  [ :re | re meaning1 do: [ :str | self assert: (re meaning2: str) ] ]
```

- 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

The screenshot shows the Dandelion web browser interface. The main content area is titled "Dandelion" and contains two sections: "All Categories" and "All Globals".

All Categories

- [CS510ap-RegularExpressions](#)

All Globals

- [ActiveEvent](#)
- [ActiveHand](#)
- [CustomEventsRegistry](#)
- [Display](#)
- [ImageImports](#)
- [Processor](#)
- [ScheduledControllers](#)
- [ScriptingSystem](#)
- [Sensor](#)
- [Smalltalk](#)
- [SourceFiles](#)
- [SystemOrganization](#)
- [TestConstants](#)
- [Transcript](#)

The left sidebar contains navigation links: "Dandelion", "Overview", "Whole Index", "All Classes", "All Categories", and "All Classes". Below these is a list of classes under "All Classes":

- [REChar](#)
- [REConcat](#)
- [REEmpty](#)
- [REStar](#)
- [REUnion](#)
- [RegularExpres](#)
- [RegularExpres](#)

At the bottom of the sidebar, there is a link for "[^top](#)" and a footer that reads "- made by [Dandelion](#) -".

Finite State Machines

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 Σ is a system $(K, \Sigma, \delta, q_0, F)$, where K is a finite, nonempty set of *states*, Σ is a finite *input alphabet*, δ 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 Σ which appear on the input tape. The interpretation of $\delta(a, a) = n$ for a

The code with NFSM

Dandelion
[Overview](#)
[Whole Index](#)

[All Classes](#)

All Categories

[CS510ap-RegularExpressions](#)

All Globals

- [ActiveEvent](#)
- [ActiveHand](#)
- [CustomEventsRegistry](#)
- [Display](#)
- [ImageImports](#)
- [Processor](#)
- [ScheduledControllers](#)
- [ScriptingSystem](#)
- [Sensor](#)
- [Smalltalk](#)
- [SourceFiles](#)
- [SystemOrganization](#)
- [TestConstants](#)
- [Transcript](#)
- [Undeclared](#)

[^top](#)

- made by Dandelion

-