
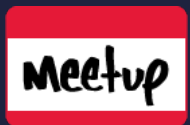


FLATMAP(OSLO) 13 MAY 2014

REGEXES, KLEENE ALGEBRAS,
AND REAL ULTIMATE POWER!

ERIK OSHEIM (@D6)

WHO AM I?

- ▶ typelevel member λ
- ▶ maintain spire and several other scala libraries
- ▶ interested in expressiveness and performance 
- ▶ hack scala code at meetup 

code at <http://github.com/non>

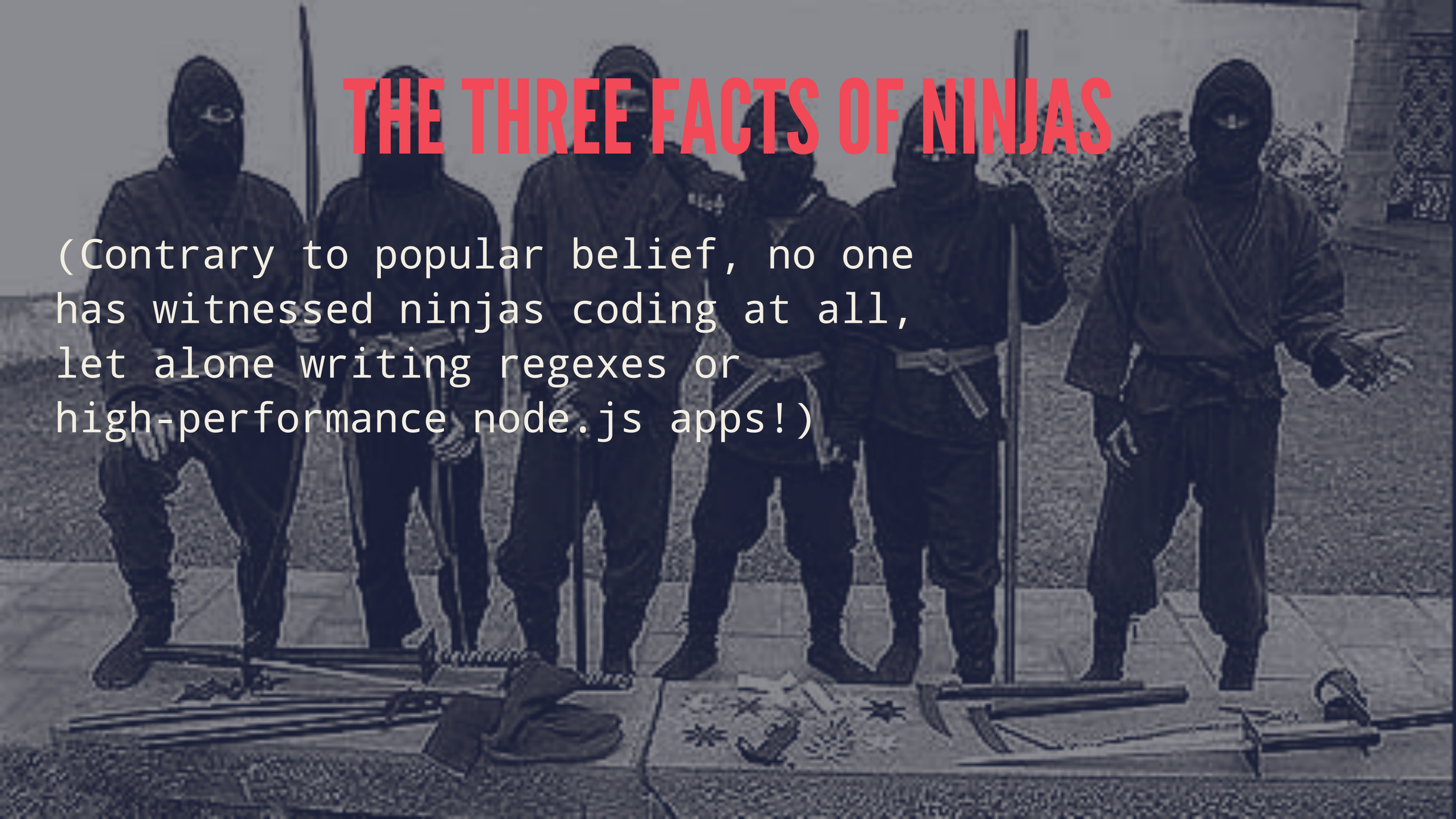
THE THREE FACTS OF REGEXES

1. Regexes are finite automaton.
2. Regexes backtrack ALL the time.
3. The purpose of a regex is to flip out and kill people.

<http://www.realultimatepower.net/index4.htm>

THE THREE FACTS OF NINJAS

(Contrary to popular belief, no one has witnessed ninjas coding at all, let alone writing regexes or high-performance node.js apps!)



OVERVIEW

regular expressions (**aka** regexps **or** regexes)

- ▶ **formalized by** Stephen Kleene (1956)
- ▶ **popularized in** unix (grep, ed, sed, awk, etc.)
 - ▶ **mainstay of modern** programming
- ▶ **provided as** syntax (e.g. perl, ruby, js, php) **or** library

EXAMPLES

positive integer

`[1-9][0-9]*`

string with escaping

`"([^\\"|\\.])*"`

source file name

`.*\.(clj|erl|hs|idr|scala|scm|sml)`

Some people, when confronted
with a problem, think "I know,
I'll use regular expressions."
Now they have two problems.

Jamie Zawinski, 1997
`alt.religion.emacs`

REGEX PROBLEMS

1. use with non-regular grammars (parsing with regexes)
2. coercing arbitrary data into byte-streams (the unix problem)
3. terse, "unreadable" regex syntax
4. extended features (the perl problem)

PERL.JPG

```
my $var = 'testing';  
$_ = 'In this string we are $var the "e" modifier.';  
  
s/(\$\w+)/$1/ee;  
  
print; # prints "In this string we are testing the "e" modifier."
```

It's easy to try to fix "regex problems" with more regexes.

This is almost always the wrong thing to do. 😞

So why is this talk about regular expressions anyway?

GOOD THINGS ABOUT REGEXES

1. composable
2. deterministic and easy to test
3. total (although can be very slow)
4. pure (side effects are impossible)

(...but any of these can be broken by "extended features".)

We will explore the power of regular expressions
by generalizing them to a more abstract concept:

Kleene algebras

Credit for this idea belongs to Russell O'Connor, who wrote "A Very General Method of Computing Shortest Paths" in *literate Haskell*.

IMPLEMENTING REGULAR EXPRESSIONS

Let's create a simple **AST** (abstract syntax tree) which can represent basic regular expressions.

We'll use sealed traits, and separate data from logic.

AST IMPLEMENTATION (1 OF 2)

```
sealed trait Expr[+A]
```

```
// matches nothing, represents  $\emptyset$  (the null set).
```

```
case object Null extends Expr[Nothing]
```

```
// matches the empty string ( $\epsilon$ ).
```

```
case object Empty extends Expr[Nothing]
```

```
// matches a literal value (a).
```

```
case class Var[A](a: A) extends Expr[A]
```


AST IMPLEMENTATION (2 OF 2)

// matches x or y , represents $(x|y)$.

```
case class Or[A](x: Expr[A], y: Expr[A]) extends Expr[A]
```

// matches x and y , represents (xy) .

```
case class Then[A](x: Expr[A], y: Expr[A]) extends Expr[A]
```

// matches zero-or-more x s, represents (x^*) .

```
case class Star[A](x: Expr[A]) extends Expr[A]
```

WEIRD REQUEST

In the next slides, we'll introduce some algebra.

In the context of regexes, let's imagine that:

$(x + y)$ means the regex $(x|y)$

$(x \cdot y)$ means the regex (xy)

(It may help to think of sum vs. product types.)

ALGEBRA DETOUR

Let's talk about algebras.

An algebra is just a set (e.g. A) together with operations on that set. For example:

$$(A + A) \rightarrow A$$

$$(-A) \rightarrow A$$

Let's define a **Rig** as an algebra with the following:

$$(A + A) \rightarrow A$$

$$(A \cdot A) \rightarrow A$$

$$0 \text{ such that } (x + 0) = x$$

$$1 \text{ such that } (x \cdot 1) = x \text{ and } (x \cdot 0) = 0$$

Both operators are associative, and **+** is commutative.

Type class implementation:

```
trait Rig[A] {  
  def zero: A  
  def one: A  
  def plus(x: A, y: A): A  
  def times(x: A, y: A): A  
}
```

(See `spire.algebra.Rig`)

By adding a **kstar** operate to our **Rig** we get a **Kleene Algebra**.
We can also define a **kplus** operator in terms of **kstar**.

Again, let's imagine that:

x.kstar means the regex x^*

x.kplus means the regex x^+

$(x.kplus = x \cdot x.kstar)$

There are four laws that our Kleene Algebra must obey.
They make can be understood in terms of our regex assumptions.

$$x^* = \varepsilon \mid xx^* = \varepsilon \mid x^*x$$

$$x \mid x = x$$

$$ax \mid x = x \Rightarrow a^*x \mid x = x$$

$$xa \mid x = x \Rightarrow xa^* \mid x = x$$

(If only the first law holds we call it a Star Rig instead.)

IMPLEMENTATION TIME!

```
trait ExprHasKleene[A] extends Kleene[A] {  
  def zero = Null  
  def one = Empty  
  
  // implementation continued...  
}
```

OPERATORS (1 OF 3)

```
// x + y is equivalent to x|y
def plus(x: Expr[A], y: Expr[A]): Expr[A] =
  (x, y) match {
    case (Null, e) => e
    case (e, Null) => e
    case (Empty, Empty) => Empty
    case (Empty, Star(e)) => Star(e)
    case (Star(e), Empty) => Star(e)
    case (e1, e2) => Or(e1, e2)
  }
```

OPERATORS (2 OF 3)

```
// x * y is equivalent to xy
def times(x: Expr[A], y: Expr[A]): Expr[A] =
  (x, y) match {
    case (Null, _) => Null
    case (_, Null) => Null
    case (Empty, e) => e
    case (e, Empty) => e
    case (e1, e2) => Then(e1, e2)
  }
```

OPERATORS (3 OF 3)

```
// x.kstar is equivalent to x*  
def kstar(x: Expr[A]): Expr[A] =  
  x match {  
    case Null => Empty  
    case Empty => Empty  
    case Star(e) => kstar(e)  
    case x => Star(x)  
  }
```

MATCHING

Let's look at some ways to use our **AST** to match input.

This is just a first pass, to be sure that our types
are **expressive enough**. We can **optimize later**.

Notice that there is no mention of **Char** or **String**.


```
def matches[A](expr: Expr[A], input: IndexedSeq[A]): Boolean = {  
  def fits(pos: Int, a: A) = pos < input.length && input(pos) == a  
  
  def look(expr: Expr[A], pos: Int): Stream[Int] =  
    expr match {  
      case Null => Stream.empty  
      case Empty => Stream(pos)  
      case Var(a) if fits(pos, a) => Stream(pos + 1)  
      case Var(_) => Stream.empty  
      case Or(x, y) => look(x, pos) ++ look(y, pos)  
      case Then(x, y) => look(x, pos).flatMap(look(y, _))  
      case Star(x) => pos #:: look(x, pos).flatMap(look(expr, _))  
    }  
  look(expr, 0).exists(_ == input.length)  
}
```

LAZY MATCHING

Instead of matching a fixed size collection,
let's try matching a **stream**.

(We'll use **Stream[A]** but any lazy,
sequential data structure would work.)

```
def smatches[A](expr: Expr[A], input: Stream[A]): Boolean = {  
  def fits(s: Stream[A], a: A) = s.headOption.map(_ == a).getOrElse(false)  
  
  def look(expr: Expr[A], input: Stream[A]): Stream[Stream[A]] =  
    expr match {  
      case Nil => Stream.empty  
      case Empty => Stream(input)  
      case Var(a) if fits(input, a) => Stream(input.tail)  
      case Var(_) => Stream.empty  
      case Or(x, y) => look(x, input) ++ look(y, input)  
      case Then(x, y) => look(x, input).flatMap(look(y, _))  
      case Star(x) => input #:: look(x, input).flatMap(look(expr, _))  
    }  
  
  Look(expr, input).exists(_.isEmpty)  
}
```

KICKING THE TIRES

```
// this assumes that our AST, our type class instance,  
// implicit operators, and matches/smatches are in scope  
def v[A](a: A) = Var(a)
```

```
// wo+(w|t)  
val p = v('w') * v('o').kplus * (v('w') + v('t'))  
p.matches("wow") // true  
p.matches("wooooow") // true  
p.matches("woot") // true  
p.matches("wood") // false
```

KICKING THE TIRES HARDER

```
// we can also match more interesting things
```

```
val bytes: Array[Byte] = ...
```

```
val header = bytes.take(8)
```

```
// we could add a Dot object to Expr[A]
```

```
val dot = v(0.toByte) + v(1.toByte) + ...
```

```
val fourBytes = dot * dot * dot * dot
```

```
val zero = v(0.toByte)
```

```
val fortyTwo = v(42.toByte)
```

```
val intel = v('I'.toByte) * v('I'.toByte) * fortyTwo * zero * fourBytes
```

```
val motorola = v('M'.toByte) * v('M'.toByte) * zero * fortyTwo * fourBytes
```

```
val png = intel + motorola
```

OBSERVATIONS

1. AST construction is ugly, but easy to improve.
2. We can name and compose regex fragments.
3. Matching generalizes to bytes, sequences or numbers, etc.
4. We can extend our AST to support richer constructs.

WHAT NOW?

So we created a novel regex implementation.

Is that it?

NO!

THE MATRIX

Let's start with this abstract definition of a square matrix.

```
case class Dim(n: Int)

trait Matrix[A] { lhs =>
  def dim: Dim
  def apply(x: Int, y: Int): A
  def map[B: ClassTag](f: A => B): Matrix[B]
  def +(rhs: Matrix[A])(implicit rig: Rig[A]): Matrix[A]
  def *(rhs: Matrix[A])(implicit rig: Rig[A]): Matrix[A]
}
```

THE MATRIX, RELOADED

We'll use `apply` to build matrix instances.

```
object Matrix {  
  def apply[A: ClassTag](f: (Int, Int) => A)(implicit dim: Dim): Matrix[A] = ...  
  
  def zero[A: Rig: ClassTag](implicit dim: Dim): Matrix[A] =  
    apply((x, y) => Rig[A].zero)  
  
  def one[A: Rig: ClassTag](implicit dim: Dim): Matrix[A] =  
    apply((x, y) => if (x == y) Rig[A].one else Rig[A].zero)  
}
```

THE DOORS OF PERCEPTION

Let's think about a matrix as a graph or state machine.
Each row represents transitions, and each operation
produces a new state machine.

$A + B$ takes a transition from A or B ($a|b$).

$A \cdot B$ takes a transition from A then from B (ab).

Using this insight we can define an algebra for matrices.

TAKE THE BLUE PILL

```
implicit def matrixHasKleene[A: Kleene: ClassTag](implicit dim: Dim) =  
  new Kleene[Matrix[A]] {  
    def zero: Matrix[A] = Matrix.zero  
    def one: Matrix[A] = Matrix.one  
    def plus(x: Matrix[A], y: Matrix[A]) = x + y  
    def times(x: Matrix[A], y: Matrix[A]) = x * y  
  
    def kstar(m: Matrix[A]) = zero + kplus(m)  
  
    def kplus(m: Matrix[A]) = ...  
  }
```

TAKE THE RED PILL

```
// this is essentially a generic implementation of
// the floyd-warshall algorithm
def kplus(m: Matrix[A]) = {
  def f(k: Int, m: Matrix[A]) = Matrix[A] { (x, y) =>
    m(x, y) + m(k, y) * m(k, k).kstar * m(x, k)
  }
  @tailrec def loop(m: Matrix[A], i: Int): Matrix[A] =
    if (i >= 0) loop(f(i, m), i - 1) else m
  loop(m, dim.n - 1)
}
```


DON'T PANIC!

A.kplus expands to $A + A * A.kplus$

$(A + AA + AAA + \dots)$

Rather than summing an infinite series, we can just compute the transitive closure, which will be equivalent under the assumptions of our Kleene[A] algebra.

This next bit is just definining tons of cool types to parameterize our matrix with. We'll go a bit fast.

 **Buckle up!** 

GRAPH TYPES

```
case class Edge(from: Int, to: Int)

object Graph {
  def apply(edges: Edge*)(implicit dim: Dim): Matrix[Boolean] = ...
}

object LabeledGraph {
  def apply(m: Matrix[Boolean])(implicit dim: Dim) =
    Matrix[Option[Edge]] { (x, y) =>
      if (m(x, y)) Some(Edge(y, x)) else None
    }
}
```

EXAMPLE GRAPH

```
val edges = List(  
  Edge(0, 1), Edge(1, 2), Edge(1, 3),  
  Edge(2, 4), Edge(3, 1), Edge(4, 2))
```

```
val example = Graph(edges:_*)(Dim(5))
```

example (adjacency)	example.kplus (t. closure)	example.kstar (reflective t. closure)
. x x x x x	x x x x x
. . x x .	. x x x x	. x x x x
. . . . x	. . x . x	. . x . x
. x x x x x	. x x x x
. . x x . x	. . x . x

LABELS AND PATHS

```
val labeled: Matrix[Edge] = LabeledGraph(example)
val expred: Matrix[Expr[Edge]] =
  labeled.map(_._map(Expr(_)).getOrElse(Nul))
```

expred:

∅	AB	∅	∅	∅
∅	∅	BC	BD	∅
∅	∅	∅	∅	CE
∅	DB	∅	∅	∅
∅	∅	EC	∅	∅

expred.kstar:

...

$A \rightarrow A$	ε
$A \rightarrow B$	$(AB \mid AB(BDDB)^*BDDB)$
$A \rightarrow C$	$AB(BDDB)^*(BC \mid BC(CEEC)^*CEEC)$
$A \rightarrow D$	$AB(BDDB)^*BD$
$A \rightarrow E$	$AB(BDDB)^*BC(CEEC)^*CE$
$B \rightarrow A$	\emptyset
$B \rightarrow B$	$(\varepsilon \mid (BDDB \mid BDDB(BDDB)^*BDDB))$
$B \rightarrow C$	$((BC \mid BC(CEEC)^*CEEC) \mid BDDB(BDDB)^*(BC \mid BC(CEEC)^*CEEC))$
$B \rightarrow D$	$(BD \mid BDDB(BDDB)^*BD)$
$B \rightarrow E$	$(BC(CEEC)^*CE \mid BDDB(BDDB)^*BC(CEEC)^*CE)$
$C \rightarrow A$	\emptyset
$C \rightarrow B$	\emptyset
$C \rightarrow C$	$(\varepsilon \mid (CEEC \mid CEEC(CEEC)^*CEEC))$
$C \rightarrow D$	\emptyset
$C \rightarrow E$	$(CE \mid CEEC(CEEC)^*CE)$
$D \rightarrow A$	\emptyset
$D \rightarrow B$	$(DB \mid DB(BDDB)^*BDDB)$
$D \rightarrow C$	$DB(BDDB)^*(BC \mid BC(CEEC)^*CEEC)$
$D \rightarrow D$	$(\varepsilon \mid DB(BDDB)^*BD)$
$D \rightarrow E$	$DB(BDDB)^*BC(CEEC)^*CE$
$E \rightarrow A$	\emptyset
$E \rightarrow B$	\emptyset
$E \rightarrow C$	$(EC \mid EC(CEEC)^*CEEC)$
$E \rightarrow D$	\emptyset
$E \rightarrow E$	$(\varepsilon \mid EC(CEEC)^*CE)$

LEAST COST PATH

```
sealed trait Weight[+A]
case class Finite[A](a: A) extends Weight[A]
case object Infinity extends Weight[Nothing]

object Weight {
  def apply[A](a: A): Weight[A] = Finite(a)
  def inf[A]: Weight[A] = Infinity
}
```


LEAST COST PATH

When talking about least cost, we'll define an algebra where

$x + y$ really means $\min(\text{cost}(x), \text{cost}(y))$

$x \cdot y$ really means $\text{cost}(x) + \text{cost}(y)$

This allows an infinite weight to function as 0.

WEIGHTS

```
implicit def WeightHasKleene[A: Order: Rig] = new Kleene[Weight[A]] {  
  def zero: Weight[A] = Infinity  
  def one: Weight[A] = Weight(Rig[A].zero)  
  def plus(x: Weight[A], y: Weight[A]): Weight[A] = (x, y) match {  
    case (Infinity, t) => t  
    case (t, Infinity) => t  
    case (Finite(a1), Finite(a2)) => Weight(a1 min a2)  
  }  
  def times(x: Weight[A], y: Weight[A]): Weight[A] = (x, y) match {  
    case (Infinity, _) | (_, Infinity) => Infinity  
    case (Finite(a1), Finite(a2)) => Weight(a1 + a2)  
  }  
  def kstar(x: Weight[A]): Weight[A] = one  
}
```

```
implicit val dim = Dim(6)
```

```
val edges = List(  
  (Edge(0, 1), 7), (Edge(0, 2), 9), (Edge(0, 5), 14),  
  (Edge(1, 2), 10), (Edge(1, 3), 15),  
  (Edge(2, 3), 11), (Edge(2, 5), 2),  
  (Edge(3, 4), 6), (Edge(4, 5), 9))
```

```
val weighted: Matrix[Weight[Int]] = ...
```

weighted:

∞	7	9	∞	∞	14
7	∞	10	15	∞	∞
9	10	∞	11	∞	2
∞	15	11	∞	6	∞
∞	∞	∞	6	∞	9
14	∞	2	∞	9	∞

weighted.kstar (least cost):

0	7	9	20	20	11
7	0	10	15	21	12
9	10	0	11	11	2
20	15	11	0	6	13
20	21	11	6	0	9
11	12	2	13	9	0

IT JUST KEEPS ON GOING...

We can combine paths and least cost to get shortest path, as well as generating an (potentially infinite) stream of all possible paths.

As long as we can define a Kleene algebra instance for a type `T`, we can use `Matrix[T].kstar` to "explore" the search space.

OTHER BENEFITS

- ▶ Generate stream of all possible matches, or a random match.
 - ▶ Compile Regex \rightarrow NFA \rightarrow DFA
 - ▶ Hopcroft DFA minimization + regex equality test.
- ▶ Determine subset/superset for two regexes (inclusion).
 - ▶ Compile-time syntax checking

EVEN MORE BENEFITS

Regular expressions can always be compiled to a **DFA**
(deterministic finite automata).

In practice many implementations simply use **backtracking**.

This can be... **bad**.

DERP.JPG

How long should a regular expression match take?

```
val s = "00000000000000000000000000000000"
val pattern = java.util.regex.Pattern.compile("(0*)*A")
val p: Expr[Char] = v('0').kstar.kstar * v('A')
val m: Dfa[Char] = p.minimize
```

```
timer { p.matches(s) } // 1.063 ms
timer { m.accept(s) } // 0.113 ms
timer { pattern.matcher(s).find() } // 27048.943 ms
```

WHERE CAN I BUY THIS?

An experimental library called **irreg** is available on Github:

<http://github.com/non/irreg>

(No packages published yet. More work is needed.)

SHORTCOMINGS / FUTURE WORK

1. wildcard (.) support
2. optimized character ranges (especially for **Dfa[A]**)
3. subgroup matching (and maybe references?)
4. DSL for working with **Char** and **String**
5. **Dfa[A]** combinators (complement, intersection, difference, ...)

BIG IDEAS

1. Finding the right abstract can be amazing.
2. Limiting the scope of an algebra may make it more useful
3. Keeping data separate from logic aids clarity.
4. There is always more to learn.
5. Another world is possible!

THAT'S ALL, FOLKS!

```
def s(xs: String): Expr[Char] =  
    xs.map(Var(_)).reduceLeft(_ * _)  
def o(e: Expr[Char]): Expr[Char] = e + Empty  
  
val e = o(s("mange ")) * o(s("tusen ")) * s("tak!")  
  
e.matches("tak!")  
e.matches("mange tak!")  
e.matches("tusen tak!")  
e.matches("mange tusen tak!")
```