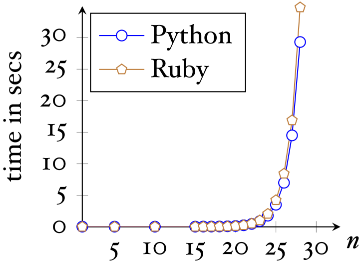
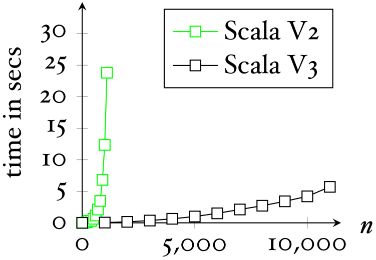
Compilers and Formal Languages (2)

Email:	christian.urban at kcl.ac.uk
Office:	N7.07 (North Wing, Bush House)
Slides:	KEATS (also homework is there)

Lets Implement an Efficient Regular Expression Matcher

Graphs: $a^{?\{n\}} \cdot a^{\{n\}}$ and strings $\underbrace{a \ldots a}_{n}$





In the handouts is a similar graph for $(a^*)^* \cdot b$ and Java 8.

Evil Regular Expressions

- Regular expression Denial of Service (ReDoS)
- Evil regular expressions
  - $a^{?\{n\}} \cdot a^{\{n\}}$
  - $(a^*)^*$
  - $([a\text{-}z]^+)^*$
  - $(a + a \cdot a)^*$
  - $(a + a^?)^*$
- sometimes also called catastrophic backtracking

Languages

- A **Language** is a set of strings, for example

$$\{[], hello, foobar, a, abc\}$$

- **Concatenation** of strings and languages

$$\begin{array}{rcl} foo \mathbin{@} bar & = & foobar \\ A \mathbin{@} B & \stackrel{\text{def}}{=} & \{s_1 \mathbin{@} s_2 \mid s_1 \in A \wedge s_2 \in B\} \end{array}$$

For example $A = \{foo, bar\}$, $B = \{a, b\}$

$$A \mathbin{@} B = \{fooa, foob, bara, barb\}$$

The Power Operation

- The ***n*th Power** of a language:

$$\begin{array}{lcl} A^0 & \stackrel{\text{def}}{=} & \{[]\} \\ A^{n+1} & \stackrel{\text{def}}{=} & A\,@\,A^n \end{array}$$

For example

$$\begin{array}{lcll} A^4 & = & A\,@\,A\,@\,A\,@\,A & (@\,\{[]\}) \\ A^1 & = & A & (@\,\{[]\}) \\ A^0 & = & \{[]\} & \end{array}$$

Homework Question

- Say $A = \{[a], [b], [c], [d]\}$.

How many strings are in A^4?

Homework Question

- Say $A = \{[a], [b], [c], [d]\}$.

How many strings are in A^4?

What if $A = \{[a], [b], [c], []\}$;
how many strings are then in A^4?

The Star Operation

- The **Kleene Star** of a language:

$$A\star \stackrel{\text{def}}{=} \bigcup_{0\le n} A^n$$

This expands to

$$A^0 \cup A^1 \cup A^2 \cup A^3 \cup A^4 \cup \ldots$$

or

$$\{[]\} \;\cup\; A \;\cup\; A\,@\,A \;\cup\; A\,@\,A\,@\,A \;\cup\; A\,@\,A\,@\,A\,@\,A \cup \ldots$$

The Meaning of a Regular Expression

$$
\begin{array}{rcl}
L(\mathbf{0}) & \stackrel{\text{def}}{=} & \{\} \\
L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{[]\} \\
L(c) & \stackrel{\text{def}}{=} & \{[c]\} \\
L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\
L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \{s_1 @ s_2 \mid s_1 \in L(r_1) \wedge s_2 \in L(r_2)\} \\
L(r^*) & \stackrel{\text{def}}{=} & (L(r))\star \quad \stackrel{\text{def}}{=} \bigcup_{0 \leq n} L(r)^n
\end{array}
$$

L is a function from regular expressions to sets of strings (languages):

$L : \text{Rexp} \Rightarrow \text{Set}[\text{String}]$

Semantic Derivative

- The **Semantic Derivative** of a language w.r.t. to a character c:

$$Der\,c\,A \stackrel{\text{def}}{=} \{s \mid c::s \in A\}$$

For $A = \{foo, bar, frak\}$ then

$$\begin{array}{lcl}
Der\,f\,A & = & \{oo, rak\}\\
Der\,b\,A & = & \{ar\}\\
Der\,a\,A & = & \{\}
\end{array}$$

Semantic Derivative

- The **Semantic Derivative** of a language w.r.t. to a character c:

$$Der\,c\,A \stackrel{\text{def}}{=} \{s \mid c::s \in A\}$$

For $A = \{foo, bar, frak\}$ then

$$\begin{aligned} Der\,f\,A &= \{oo, rak\} \\ Der\,b\,A &= \{ar\} \\ Der\,a\,A &= \{\} \end{aligned}$$

We can extend this definition to strings

$$Ders\,s\,A = \{s' \mid s@s' \in A\}$$

The Specification for Matching

A regular expression r matches a string s provided

$$s \in L(r)$$

...and the point of the this lecture is to decide this problem as fast as possible (unlike Python, Ruby, Java etc)

Regular Expressions

Their inductive definition:

$$
\begin{array}{rcll}
r & ::= & \mathbf{0} & \text{nothing} \\
 & | & \mathbf{1} & \text{empty string / "" / []} \\
 & | & c & \text{single character} \\
 & | & r_1 \cdot r_2 & \text{sequence} \\
 & | & r_1 + r_2 & \text{alternative / choice} \\
 & | & r^* & \text{star (zero or more)}
\end{array}
$$

Th

```scala
abstract class Rexp
case object ZERO extends Rexp
case object ONE extends Rexp
case class CHAR(c: Char) extends Rexp
case class ALT(r1: Rexp, r2: Rexp) extends Rexp
case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
case class STAR(r: Rexp) extends Rexp
```

r	$::=$	$\mathbf{0}$	nothing
	$\mid$	$\mathbf{1}$	empty string / "" / []
	$\mid$	c	single character
	$\mid$	$r_1 \cdot r_2$	sequence
	$\mid$	$r_1 + r_2$	alternative / choice
	$\mid$	r^*	star (zero or more)

When Are Two Regular Expressions Equivalent?

$$r_1 \equiv r_2 \quad \stackrel{\text{def}}{=} \quad L(r_1) = L(r_2)$$

Concrete Equivalences

$$\begin{array}{rcl}
(a+b)+c & \equiv & a+(b+c) \\
a+a & \equiv & a \\
a+b & \equiv & b+a \\
(a\cdot b)\cdot c & \equiv & a\cdot(b\cdot c) \\
c\cdot(a+b) & \equiv & (c\cdot a)+(c\cdot b)
\end{array}$$

Concrete Equivalences

$$
\begin{array}{rcl}
(a+b)+c & \equiv & a+(b+c) \\
a+a & \equiv & a \\
a+b & \equiv & b+a \\
(a\cdot b)\cdot c & \equiv & a\cdot(b\cdot c) \\
c\cdot(a+b) & \equiv & (c\cdot a)+(c\cdot b)
\end{array}
$$

$$
\begin{array}{rcl}
a\cdot a & \not\equiv & a \\
a+(b\cdot c) & \not\equiv & (a+b)\cdot(a+c)
\end{array}
$$

Corner Cases

$$
\begin{array}{rcl}
a \cdot \mathbf{0} & \not\equiv & a \\
a + \mathbf{1} & \not\equiv & a \\
\mathbf{1} & \equiv & \mathbf{0}^* \\
\mathbf{1}^* & \equiv & \mathbf{1} \\
\mathbf{0}^* & \not\equiv & \mathbf{0}
\end{array}
$$

Simplification Rules

$$
\begin{array}{rcl}
r + \mathbf{0} & \equiv & r \\
\mathbf{0} + r & \equiv & r \\
r \cdot \mathbf{1} & \equiv & r \\
\mathbf{1} \cdot r & \equiv & r \\
r \cdot \mathbf{0} & \equiv & \mathbf{0} \\
\mathbf{0} \cdot r & \equiv & \mathbf{0} \\
r + r & \equiv & r
\end{array}
$$

A Matching Algorithm (1)

...whether a regular expression can match the empty string:

$$\begin{array}{lcl}
nullable(\mathbf{0}) & \stackrel{\text{def}}{=} & false\\
nullable(\mathbf{1}) & \stackrel{\text{def}}{=} & true\\
nullable(c) & \stackrel{\text{def}}{=} & false\\
nullable(r_1 + r_2) & \stackrel{\text{def}}{=} & nullable(r_1) \vee nullable(r_2)\\
nullable(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & nullable(r_1) \wedge nullable(r_2)\\
nullable(r^*) & \stackrel{\text{def}}{=} & true
\end{array}$$

The Derivative of a Rexp

If r matches the string $c::s$, what is a regular expression that matches just s?

$der\,c\,r$ gives the answer, Brzozowski 1964

The Derivative of a Rexp

$$
\begin{array}{lcl}
der\,c\,(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0}\\
der\,c\,(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{0}\\
der\,c\,(d) & \stackrel{\text{def}}{=} & \text{if } c = d \text{ then } \mathbf{1} \text{ else } \mathbf{0}\\
der\,c\,(r_1 + r_2) & \stackrel{\text{def}}{=} & der\,c\,r_1 + der\,c\,r_2\\
der\,c\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1)\\
 & & \text{then } (der\,c\,r_1) \cdot r_2 + der\,c\,r_2\\
 & & \text{else } (der\,c\,r_1) \cdot r_2\\
der\,c\,(r^*) & \stackrel{\text{def}}{=} & (der\,c\,r) \cdot (r^*)
\end{array}
$$

The Derivative of a Rexp

$$\begin{array}{lcl}
der\,c\,(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0}\\
der\,c\,(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{0}\\
der\,c\,(d) & \stackrel{\text{def}}{=} & \text{if } c = d \text{ then } \mathbf{1} \text{ else } \mathbf{0}\\
der\,c\,(r_1 + r_2) & \stackrel{\text{def}}{=} & der\,c\,r_1 + der\,c\,r_2\\
der\,c\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1)\\
 & & \text{then } (der\,c\,r_1) \cdot r_2 + der\,c\,r_2\\
 & & \text{else } (der\,c\,r_1) \cdot r_2\\
der\,c\,(r^*) & \stackrel{\text{def}}{=} & (der\,c\,r) \cdot (r^*)\\
\\
ders\,[]\,r & \stackrel{\text{def}}{=} & r\\
ders\,(c::s)\,r & \stackrel{\text{def}}{=} & ders\,s\,(der\,c\,r)
\end{array}$$

Examples

Given $r \stackrel{\text{def}}{=} ((a \cdot b) + b)^*$ what is

$$
\begin{aligned}
der\, a\, r &= ?\\
der\, b\, r &= ?\\
der\, c\, r &= ?
\end{aligned}
$$

The Brzozowski Algorithm

$$matches\, r\, s \stackrel{\text{def}}{=} nullable(ders\ s\ r)$$

Brzozowski: An Example

Does r_1 match abc?

Step 1:	build derivative of a and r_1	$(r_2 = der\ a\ r_1)$
Step 2:	build derivative of b and r_2	$(r_3 = der\ b\ r_2)$
Step 3:	build derivative of c and r_3	$(r_4 = der\ c\ r_3)$
Step 4:	the string is exhausted: test whether r_4 can recognise the empty string	$(nullable(r_4))$
Output:	result of the test $\Rightarrow$ $true$ or $false$	

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r_1 then

1. $Der\,a\,(L(r_1))$

The Idea of the Algorithm

If we want to recognise the string abc with regular expression r_1 then

1. $Der\,a\,(L(r_1))$
2. $Der\,b\,(Der\,a\,(L(r_1)))$

The Idea of the Algorithm

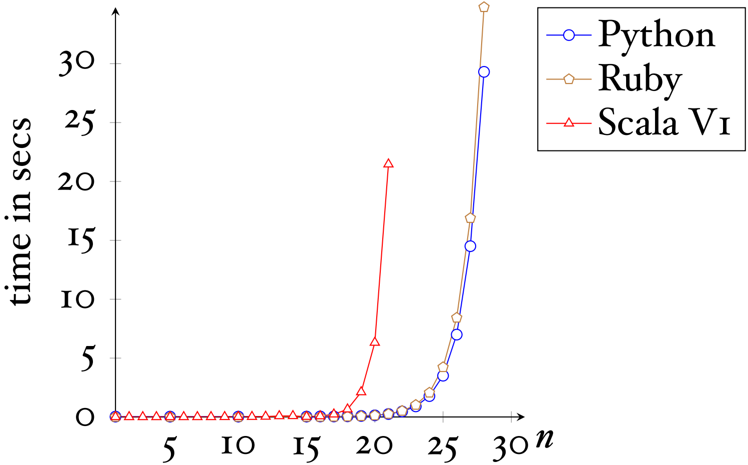
If we want to recognise the string abc with regular expression r_1 then

1. $Der\, a\, (L(r_1))$
2. $Der\, b\, (Der\, a\, (L(r_1)))$
3. $Der\, c\, (Der\, b\, (Der\, a\, (L(r_1))))$

4. finally we test whether the empty string is in this set; same for $Ders\, abc\, (L(r_1))$.

The matching algorithm works similarly, just over regular expressions instead of sets.

Oops... $a^{?\{n\}} \cdot a^{\{n\}}$



A Problem

We represented the "n-times" $a^{\{n\}}$ as a sequence regular expression:

$$
\begin{array}{rl}
1: & a \\
2: & a \cdot a \\
3: & a \cdot a \cdot a \\
 & \ldots \\
13: & a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \\
 & \ldots \\
20: &
\end{array}
$$

This problem is aggravated with $a^?$ being represented as $a + \mathbf{1}$.

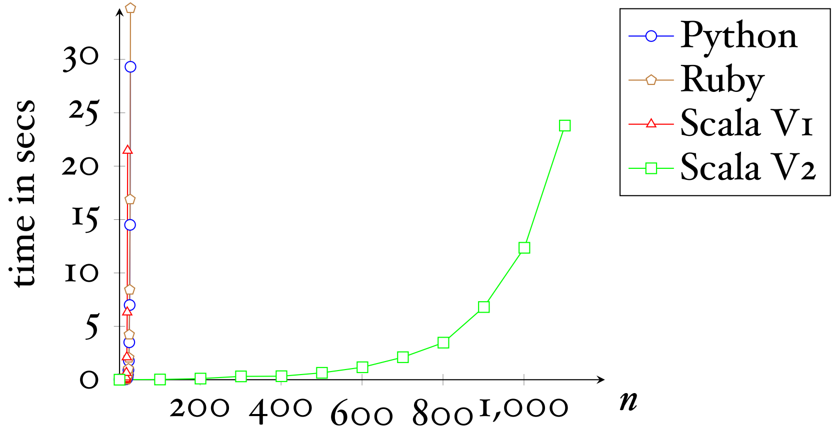
Solving the Problem

What happens if we extend our regular expressions with explicit constructors

$$\begin{array}{rcl} r & ::= & \ldots \\ & | & r^{\{n\}} \\ & | & r^{?} \end{array}$$

What is their meaning?
What are the cases for *nullable* and *der*?

Brzozowski: $a^{?\{n\}} \cdot a^{\{n\}}$



Examples

Recall the example of $r \stackrel{\text{def}}{=} ((a \cdot b) + b)^*$ with

$$
\begin{aligned}
der\,a\,r &= ((\mathbf{1} \cdot b) + \mathbf{0}) \cdot r \\
der\,b\,r &= ((\mathbf{0} \cdot b) + \mathbf{1}) \cdot r \\
der\,c\,r &= ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot r
\end{aligned}
$$

What are these regular expressions equivalent to?

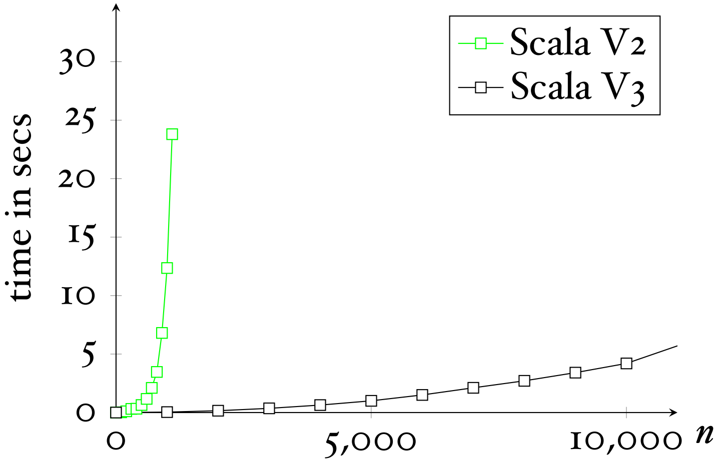
Simplification Rules

$$
\begin{array}{rcl}
r + \mathbf{0} & \Rightarrow & r \\
\mathbf{0} + r & \Rightarrow & r \\
r \cdot \mathbf{1} & \Rightarrow & r \\
\mathbf{1} \cdot r & \Rightarrow & r \\
r \cdot \mathbf{0} & \Rightarrow & \mathbf{0} \\
\mathbf{0} \cdot r & \Rightarrow & \mathbf{0} \\
r + r & \Rightarrow & r
\end{array}
$$

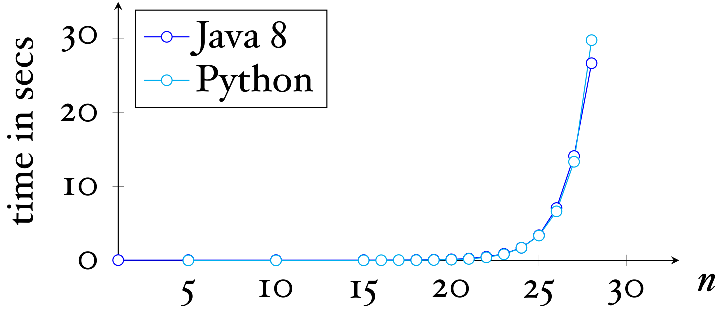
```
def ders(s: List[Char], r: Rexp) : Rexp = s match {
  case Nil => r
  case c::s => ders(s, simp(der(c, r)))
}
```

```scala
def simp(r: Rexp) : Rexp = r match {
  case ALT(r1, r2) => {
    (simp(r1), simp(r2)) match {
      case (ZERO, r2s) => r2s
      case (r1s, ZERO) => r1s
      case (r1s, r2s) =>
        if (r1s == r2s) r1s else ALT(r1s, r2s)
    }
  }
  case SEQ(r1, r2) => {
    (simp(r1), simp(r2)) match {
      case (ZERO, _) => ZERO
      case (_, ZERO) => ZERO
      case (ONE, r2s) => r2s
      case (r1s, ONE) => r1s
      case (r1s, r2s) => SEQ(r1s, r2s)
    }
  }
  case r => r
}
```

Brzozowski: $a^{?\{n\}} \cdot a^{\{n\}}$

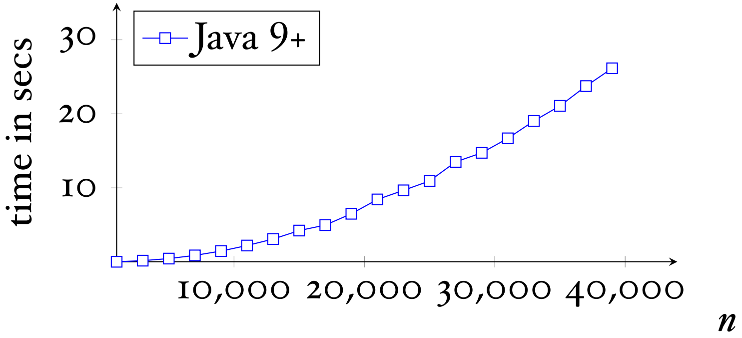


Another Example in Java 8 and Python



Regex: $(a^*)^* \cdot b$
Strings of the form $\underbrace{a \ldots a}_{n}$

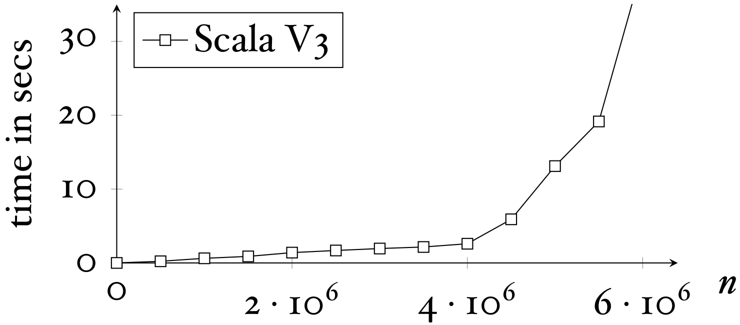
Same Example in Java 9+



Regex: $(a^*)^* \cdot b$

Strings of the form $\underbrace{a \ldots a}_{n}$

and with Brzozowski



Regex: $(a^*)^* \cdot b$

Strings of the form $\underbrace{a \ldots a}_{n}$

What is good about this Alg.

- extends to most regular expressions, for example $\sim r$
- is easy to implement in a functional language
- the algorithm is already quite old; there is still work to be done to use it as a tokenizer (that is relatively new work)
- we can prove its correctness...

Proofs about Rexps

Remember their inductive definition:

$$
\begin{array}{rcl}
r & ::= & \mathbf{0} \\
  & \mid & \mathbf{1} \\
  & \mid & c \\
  & \mid & r_1 \cdot r_2 \\
  & \mid & r_1 + r_2 \\
  & \mid & r^*
\end{array}
$$

If we want to prove something, say a property $P(r)$, for all regular expressions r then ...

Proofs about Rexp (2)

- P holds for $\mathbf{0}$, $\mathbf{1}$ and c
- P holds for $r_1 + r_2$ under the assumption that P already holds for r_1 and r_2.
- P holds for $r_1 \cdot r_2$ under the assumption that P already holds for r_1 and r_2.
- P holds for r^* under the assumption that P already holds for r.

Proofs about Rexp (3)

Assume $P(r)$ is the property:

$$\mathit{nullable}(r) \text{ if and only if } [] \in L(r)$$

Proofs about Rexp (4)

$$
\begin{array}{rcl}
rev(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
rev(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{1} \\
rev(c) & \stackrel{\text{def}}{=} & c \\
rev(r_1 + r_2) & \stackrel{\text{def}}{=} & rev(r_1) + rev(r_2) \\
rev(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & rev(r_2) \cdot rev(r_1) \\
rev(r^*) & \stackrel{\text{def}}{=} & rev(r)^*
\end{array}
$$

We can prove

$$L(rev(r)) = \{s^{-1} \mid s \in L(r)\}$$

by induction on r.

Correctness Proof for our Matcher

- We started from

$$
\begin{array}{rl}
& s \in L(r) \\
\Leftrightarrow & [] \in Ders\, s\, (L(r))
\end{array}
$$

Correctness Proof for our Matcher

- We started from

$$
\begin{array}{rl}
 & s \in L(r) \\
\Leftrightarrow & [] \in Ders\, s\, (L(r))
\end{array}
$$

- if we can show $Ders\, s\, (L(r)) = L(ders\, s\, r)$ we have

$$
\begin{array}{rl}
\Leftrightarrow & [] \in L(ders\, s\, r) \\
\Leftrightarrow & nullable(ders\, s\, r) \\
\stackrel{\text{def}}{=} & matches\, s\, r
\end{array}
$$

Proofs about Rexp (5)

Let $Der\,c\,A$ be the set defined as

$$Der\,c\,A \stackrel{\text{def}}{=} \{s \mid c::s \in A\}$$

We can prove

$$L(der\,c\,r) = Der\,c\,(L(r))$$

by induction on r.

Proofs about Strings

If we want to prove something, say a property $P(s)$, for all strings s then ...

- P holds for the empty string, and
- P holds for the string $c::s$ under the assumption that P already holds for s

Proofs about Strings (2)

We can then prove

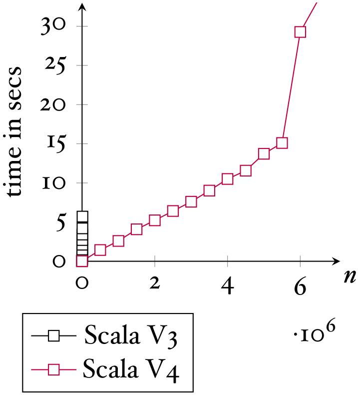
$$Ders\,s\,(L(r)) = L(ders\,s\,r)$$

We can finally prove

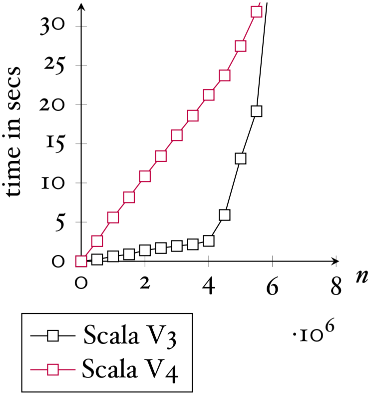
$$matches\,s\,r \text{ if and only if } s \in L(r)$$

Epilogue

Graph: $a^{?\{n\}} \cdot a^{\{n\}}$



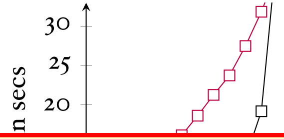
Graph: $(a^*)^* \cdot b$



Epilogue

Graph: $a^{?\{n\}} \cdot a^{\{n\}}$

Graph: $(a^*)^* \cdot b$




```
def ders2(s: List[Char], r: Rexp) : Rexp = (s, r) match {
  case (Nil, r) => r
  case (s, ZERO) => ZERO
  case (s, ONE) => if (s == Nil) ONE else ZERO
  case (s, CHAR(c)) => if (s == List(c)) ONE else
                       if (s == Nil) CHAR(c) else ZERO
  case (s, ALT(r1, r2)) => ALT(ders2(s, r2), ders2(s, r2))
  case (c::s, r) => ders2(s, simp(der(c, r)))
}
```