

Compilers and Formal Languages

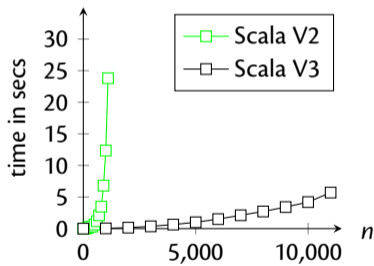
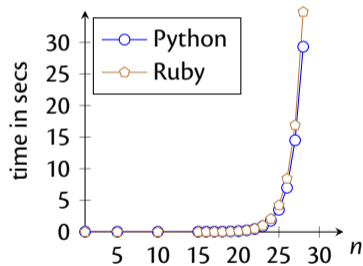
Email: christian.urban at kcl.ac.uk

Slides & Progs: KEATS (also homework is there)

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

Let's Implement an Efficient Regular Expression Matcher

Graphs: $a^{?{n}} \cdot a^{n}$ and strings $\underbrace{a \dots a}_n$



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

(Basic) Regular Expressions

Their inductive definition:

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

When Are Two Regular Expressions Equivalent?

Two regular expressions r_1 and r_2 are **equivalent** provided:

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

Some Concrete Equivalences

$$(a + b) + c \equiv a + (b + c)$$

$$a + a \equiv a$$

$$a + b \equiv b + a$$

$$(a \cdot b) \cdot c \equiv a \cdot (b \cdot c)$$

$$c \cdot (a + b) \equiv (c \cdot a) + (c \cdot b)$$

Some Concrete Equivalences

$$(a + b) + c \equiv a + (b + c)$$

$$a + a \equiv a$$

$$a + b \equiv b + a$$

$$(a \cdot b) \cdot c \equiv a \cdot (b \cdot c)$$

$$c \cdot (a + b) \equiv (c \cdot a) + (c \cdot b)$$

$$a \cdot a \not\equiv a$$

$$a + (b \cdot c) \not\equiv (a + b) \cdot (a + c)$$

Some Corner Cases

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

Some Simplification Rules

$$r + 0 \equiv r$$

$$0 + r \equiv r$$

$$r \cdot 1 \equiv r$$

$$1 \cdot r \equiv r$$

$$r \cdot 0 \equiv 0$$

$$0 \cdot r \equiv 0$$

$$r + r \equiv r$$

Simplification Example

$$\begin{aligned}((\mathbf{1} \cdot b) + \mathbf{0}) \cdot r &\Rightarrow ((\underline{\mathbf{1} \cdot b}) + \mathbf{0}) \cdot r \\ &= (\underline{b + \mathbf{0}}) \cdot r \\ &= b \cdot r\end{aligned}$$

Simplification Example

$$\begin{aligned}((\mathbf{0} \cdot b) + \mathbf{0}) \cdot r &\Rightarrow ((\underline{\mathbf{0} \cdot b}) + \mathbf{0}) \cdot r \\ &= (\underline{\mathbf{0} + \mathbf{0}}) \cdot r \\ &= \mathbf{0} \cdot r \\ &= \mathbf{0}\end{aligned}$$

Semantic Derivative

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

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

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

$$\text{Der } f A = \{\text{oo}, \text{rak}\}$$

$$\text{Der } b A = \{\text{ar}\}$$

$$\text{Der } a A = \{\}$$

Semantic Derivative

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

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

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

$$\text{Der } f A = \{\text{oo}, \text{rak}\}$$

$$\text{Der } b A = \{\text{ar}\}$$

$$\text{Der } a A = \{\}$$

We can extend this definition to strings

$$\text{Der } s A = \{s' \mid s @ s' \in A\}$$

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)

Brzowski's Algorithm (1)

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

$$\text{nullable}(\mathbf{0}) \stackrel{\text{def}}{=} \text{false}$$

$$\text{nullable}(\mathbf{1}) \stackrel{\text{def}}{=} \text{true}$$

$$\text{nullable}(c) \stackrel{\text{def}}{=} \text{false}$$

$$\text{nullable}(r_1 + r_2) \stackrel{\text{def}}{=} \text{nullable}(r_1) \vee \text{nullable}(r_2)$$

$$\text{nullable}(r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{nullable}(r_1) \wedge \text{nullable}(r_2)$$

$$\text{nullable}(r^*) \stackrel{\text{def}}{=} \text{true}$$

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

$$\text{der } c(0) \stackrel{\text{def}}{=} 0$$

$$\text{der } c(1) \stackrel{\text{def}}{=} 0$$

$$\text{der } c(d) \stackrel{\text{def}}{=} \text{if } c = d \text{ then } 1 \text{ else } 0$$

$$\text{der } c(r_1 + r_2) \stackrel{\text{def}}{=} \text{der } c r_1 + \text{der } c r_2$$

$$\text{der } c(r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{if nullable}(r_1) \\ \text{then } (\text{der } c r_1) \cdot r_2 + \text{der } c r_2 \\ \text{else } (\text{der } c r_1) \cdot r_2$$

$$\text{der } c(r^*) \stackrel{\text{def}}{=} (\text{der } c r) \cdot (r^*)$$

The Derivative of a Rexp

$$\text{der } c \text{ (0)} \stackrel{\text{def}}{=} \mathbf{0}$$

$$\text{der } c \text{ (1)} \stackrel{\text{def}}{=} \mathbf{0}$$

$$\text{der } c \text{ (} d \text{)} \stackrel{\text{def}}{=} \text{if } c = d \text{ then } \mathbf{1} \text{ else } \mathbf{0}$$

$$\text{der } c \text{ (} r_1 + r_2 \text{)} \stackrel{\text{def}}{=} \text{der } c \text{ } r_1 + \text{der } c \text{ } r_2$$

$$\text{der } c \text{ (} r_1 \cdot r_2 \text{)} \stackrel{\text{def}}{=} \text{if nullable}(r_1) \\ \text{then } (\text{der } c \text{ } r_1) \cdot r_2 + \text{der } c \text{ } r_2 \\ \text{else } (\text{der } c \text{ } r_1) \cdot r_2$$

$$\text{der } c \text{ (} r^* \text{)} \stackrel{\text{def}}{=} (\text{der } c \text{ } r) \cdot (r^*)$$

$$\text{ders } [] \text{ } r \stackrel{\text{def}}{=} r$$

$$\text{ders } (c :: s) \text{ } r \stackrel{\text{def}}{=} \text{ders } s \text{ (der } c \text{ } r)$$

Examples

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

$\text{der } a \ r = ?$

$\text{der } b \ r = ?$

$\text{der } c \ r = ?$

Derivative Example

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

$$\begin{aligned} \text{der } a \ ((a \cdot b) + b)^* &\Rightarrow \text{der } a \ \underline{((a \cdot b) + b)^*} \\ &= (\text{der } a \ \underline{((a \cdot b) + b)}) \cdot r \\ &= ((\text{der } a \ \underline{a \cdot b}) + (\text{der } a \ b)) \cdot r \\ &= (((\text{der } a \ \underline{a}) \cdot b) + (\text{der } a \ b)) \cdot r \\ &= ((\mathbf{1} \cdot b) + (\text{der } a \ \underline{b})) \cdot r \\ &= ((\mathbf{1} \cdot b) + \mathbf{0}) \cdot r \end{aligned}$$

The Brzozowski Algorithm

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

Brzozowski: An Example

Does r_1 match abc ?

Step 1: build derivative of a and r_1 ($r_2 = \text{der } a \ r_1$)

Step 2: build derivative of b and r_2 ($r_3 = \text{der } b \ r_2$)

Step 3: build derivative of c and r_3 ($r_4 = \text{der } c \ r_3$)

Step 4: the string is exhausted: ($\text{nullable}(r_4)$)

test whether r_4 can recognise
the empty string

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

$$\text{Der } a(L(r_1))$$

The Idea of the Algorithm

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

$Der a (L(r_1))$

$Der b (Der a (L(r_1)))$

The Idea of the Algorithm

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

$Der a (L(r_1))$

$Der b (Der a (L(r_1)))$

$Der c (Der b (Der a (L(r_1))))$

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

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

The Idea with Derivatives

Input: string *abc* and regular expression *r*

der a r

der b (der a r)

der c (der b (der a r))

The Idea with Derivatives

Input: string *abc* and regular expression *r*

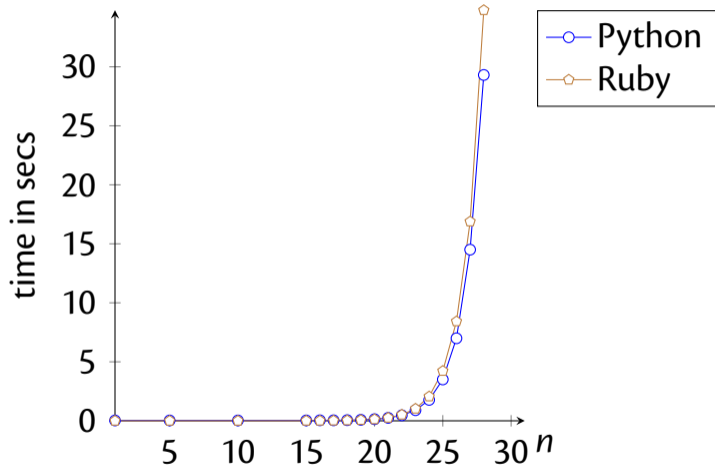
der a r

der b (der a r)

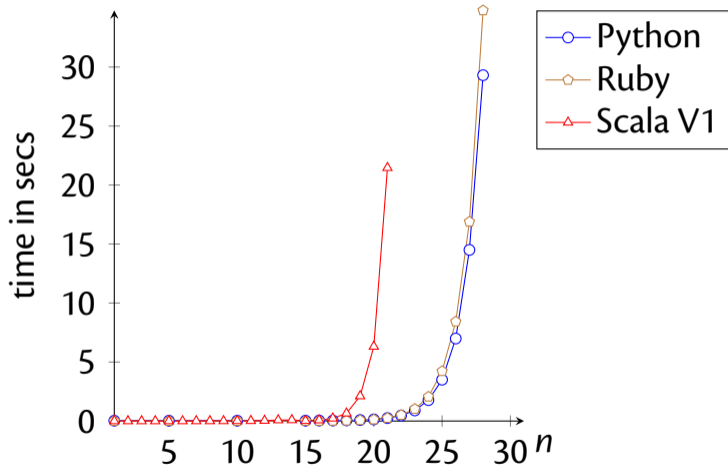
der c (der b (der a r))

finally check whether the last regular expression can match the empty string

$$a^{\{n\}} \cdot a^{\{n\}}$$



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



A Problem

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

0: **1**

1: a

2: $a \cdot a$

3: $a \cdot a \cdot a$

...

13: $a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$

...

20:

This problem is aggravated with $a^?$ being represented as $a + 1$.

Solving the Problem

What happens if we extend our regular expressions with explicit constructors

$$r ::= \dots$$
$$| r^{\{n\}}$$
$$| r^?$$

What is their meaning?

What are the cases for *nullable* and *der*?

der for n -times

Case $n = 2$ and $r \cdot r$:

$$\begin{aligned} \text{der } c(r \cdot r) &\stackrel{\text{def}}{=} \text{if } \text{nullable}(r) \\ &\quad \text{then } (\text{der } c r) \cdot r + \text{der } c r \\ &\quad \text{else } (\text{der } c r) \cdot r \end{aligned}$$

der for n -times

Case $n = 2$ and $r \cdot r$:

$$\begin{aligned} \text{der } c(r \cdot r) &\stackrel{\text{def}}{=} \text{if } \text{nullable}(r) \\ &\quad \text{then } (\text{der } c r) \cdot r + \text{der } c r \\ &\quad \text{else } (\text{der } c r) \cdot r \end{aligned}$$

$$\begin{aligned} \text{my claim} &\equiv (\text{der } c r) \cdot r \\ \text{(in this case)} & \end{aligned}$$

der for n -times

Case $n = 2$ and $r \cdot r$:

$$\begin{aligned} \text{der } c(r \cdot r) &\stackrel{\text{def}}{=} \text{if } \text{nullable}(r) \\ &\text{then } (\text{der } cr) \cdot r + \text{der } cr \\ &\text{else } (\text{der } cr) \cdot r \end{aligned}$$

$$\begin{aligned} \text{my claim} &\equiv (\text{der } cr) \cdot r \\ \text{(in this case)} & \end{aligned}$$

We know *nullable*(*r*) holds!

We know *nullable*(*r*) holds!

$$(der\ c\ r) \cdot r + der\ c\ r$$

We know *nullable*(*r*) holds!

$$(der\ cr) \cdot r + der\ cr \equiv (der\ cr) \cdot r + (der\ cr) \cdot \mathbf{1}$$

We know *nullable*(*r*) holds!

$$\begin{aligned}(\text{der } cr) \cdot r + \text{der } cr &\equiv (\text{der } cr) \cdot r + (\text{der } cr) \cdot \mathbf{1} \\ &\equiv (\text{der } cr) \cdot (r + \mathbf{1})\end{aligned}$$

We know *nullable*(*r*) holds!

$$\begin{aligned}(\text{der } c r) \cdot r + \text{der } c r &\equiv (\text{der } c r) \cdot r + (\text{der } c r) \cdot \mathbf{1} \\ &\equiv (\text{der } c r) \cdot (r + \mathbf{1}) \\ &\equiv (\text{der } c r) \cdot r \\ &\quad \text{(remember } r \text{ is nullable)}\end{aligned}$$

We know *nullable*(*r*) holds!

$$\begin{aligned}(\text{der } c \ r) \cdot r + \text{der } c \ r &\equiv (\text{der } c \ r) \cdot r + (\text{der } c \ r) \cdot \mathbf{1} \\ &\equiv (\text{der } c \ r) \cdot (r + \mathbf{1}) \\ &\equiv (\text{der } c \ r) \cdot r \\ &\quad \text{(remember } r \text{ is nullable)}\end{aligned}$$

$$\text{der } c \ (r \cdot r) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } \text{nullable}(r) \\ \text{then } (\text{der } c \ r) \cdot r + \text{der } c \ r \\ \text{else } (\text{der } c \ r) \cdot r \end{array}$$

We know *nullable*(*r*) holds!

$$\begin{aligned}(\text{der } c r) \cdot r + \text{der } c r &\equiv (\text{der } c r) \cdot r + (\text{der } c r) \cdot \mathbf{1} \\ &\equiv (\text{der } c r) \cdot (r + \mathbf{1}) \\ &\equiv (\text{der } c r) \cdot r \\ &\quad \text{(remember } r \text{ is nullable)}\end{aligned}$$

$$\text{der } c (r \cdot r) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } \text{nullable}(r) \\ \text{then } (\text{der } c r) \cdot r \\ \text{else } (\text{der } c r) \cdot r \end{array}$$

We know *nullable*(*r*) holds!

$$\begin{aligned}(\text{der } c r) \cdot r + \text{der } c r &\equiv (\text{der } c r) \cdot r + (\text{der } c r) \cdot \mathbf{1} \\ &\equiv (\text{der } c r) \cdot (r + \mathbf{1}) \\ &\equiv (\text{der } c r) \cdot r \\ &\quad \text{(remember } r \text{ is nullable)}\end{aligned}$$

$$\text{der } c (r \cdot r) \stackrel{\text{def}}{=} (\text{der } c r) \cdot r$$

	$r\{n\}$	der
$n = 0:$	1	0
$n = 1:$	r	$(der\ r)$
$n = 2:$	$r \cdot r$	$(der\ r) \cdot r$
$n = 3:$	$r \cdot r \cdot r$???
	\vdots	

	$r\{n\}$	der
$n = 0:$	1	0
$n = 1:$	r	$(der\ c\ r)$
$n = 2:$	$r \cdot r$	$(der\ c\ r) \cdot r$
$n = 3:$	$r \cdot r \cdot r$	$(der\ c\ r) \cdot r \cdot r$
	\vdots	

	$r\{n\}$	der
$n = 0:$	1	0
$n = 1:$	r	$(der\ c\ r)$
$n = 2:$	$r \cdot r$	$(der\ c\ r) \cdot r$
$n = 3:$	$r \cdot r \cdot r$	$(der\ c\ r) \cdot r \cdot r$
	\vdots	

$nullable(r\{n\}) \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } true \text{ else } nullable(r)$

$der\ c\ (r\{n\}) \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then } \mathbf{0} \text{ else } (der\ c\ r) \cdot r\{n - 1\}$

$derc(r \cdot r \cdot r) \stackrel{\text{def}}{=} \begin{cases} \text{if } nullable(r) \\ \text{then } (derc\ r) \cdot r \cdot r + derc(r \cdot r) \\ \text{else } (derc\ r) \cdot r \cdot r \end{cases}$

$derc(r \cdot r \cdot r) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } nullable(r) \\ \text{then } (derc\ r) \cdot r \cdot r + (derc\ r) \cdot r \\ \text{else } (derc\ r) \cdot r \cdot r \end{array}$

$derc(r \cdot r \cdot r) \stackrel{\text{def}}{=} \begin{cases} \text{if } nullable(r) \\ \text{then } (derc\ r) \cdot (r \cdot r + r) \\ \text{else } (derc\ r) \cdot r \cdot r \end{cases}$

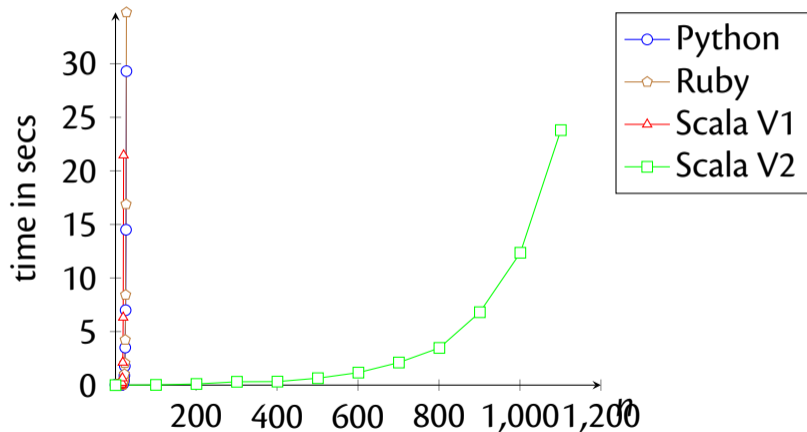
$derc(r \cdot r \cdot r) \stackrel{\text{def}}{=} \text{if } nullable(r)$
then $(derc r) \cdot (r \cdot (r + \mathbf{1}))$
else $(derc r) \cdot r \cdot r$

$derc(r \cdot r \cdot r) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } nullable(r) \\ \text{then } (derc\ r) \cdot (r \cdot r) \\ \text{else } (derc\ r) \cdot r \cdot r \end{array}$

$derc(r \cdot r \cdot r) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } nullable(r) \\ \text{then } (derc\ r) \cdot r \cdot r \\ \text{else } (derc\ r) \cdot r \cdot r \end{array}$

$$\text{der } c(r \cdot r \cdot r) \stackrel{\text{def}}{=} (\text{der } cr) \cdot r \cdot r$$

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



Examples

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

$$\text{der } a r = ((\mathbf{1} \cdot b) + \mathbf{0}) \cdot r$$

$$\text{der } b r = ((\mathbf{0} \cdot b) + \mathbf{1}) \cdot r$$

$$\text{der } c r = ((\mathbf{0} \cdot b) + \mathbf{0}) \cdot r$$

What are these regular expressions equivalent to?

Simplification Rules

$$r + 0 \Rightarrow r$$

$$0 + r \Rightarrow r$$

$$r \cdot 1 \Rightarrow r$$

$$1 \cdot r \Rightarrow r$$

$$r \cdot 0 \Rightarrow 0$$

$$0 \cdot r \Rightarrow 0$$

$$r + r \Rightarrow r$$

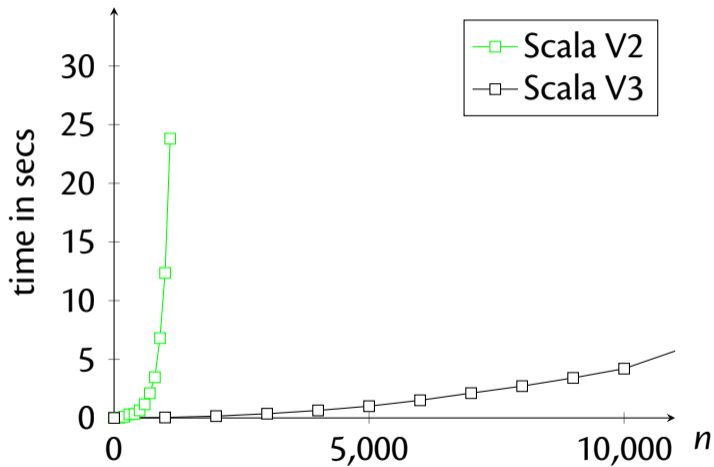
```
def ders(s: List[Char], r: Rexp) : Rexp = s match {  
  case Nil => r  
  case c::s => ders(s, simp(der(c, r)))  
}
```

```

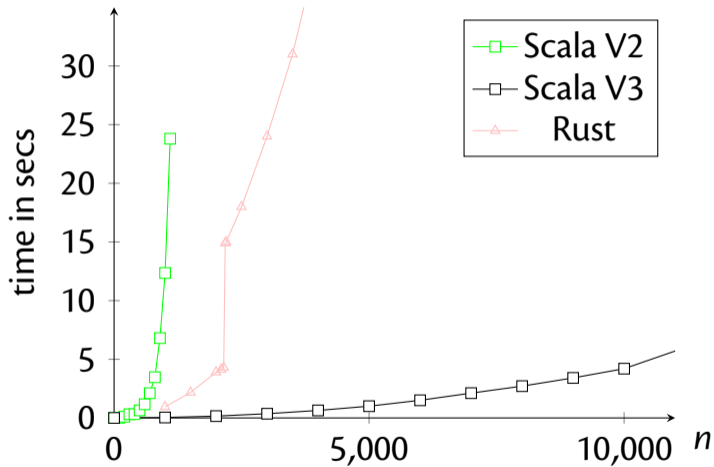
def simp(r: Rexp) : Rexp = r match {
  case ALT(r1, r2) => {
    (simp(r1), simp(r2)) match {
      case (ZERO, r2s) => r2s
      case (r1s, ZERO) => r1s
      case (r1s, r2s) =>
        if (r1s == r2s) r1s else ALT(r1s, r2s)
    }
  }
  case SEQ(r1, r2) => {
    (simp(r1), simp(r2)) match {
      case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

```

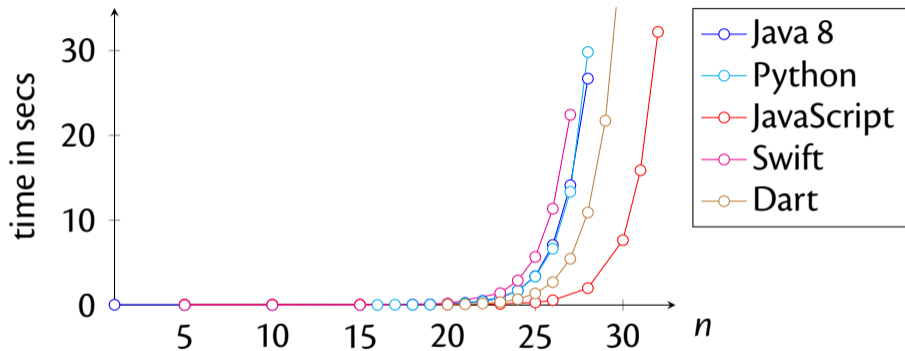
Brzozowski: $a^{\{n\}} \cdot a^{\{n\}}$



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



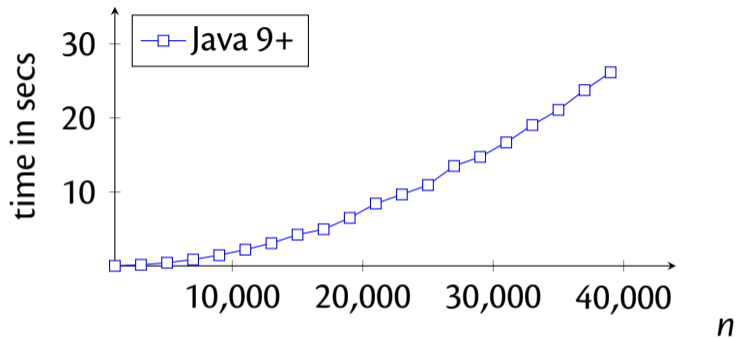
Another Example $(a^*)^* \cdot b$



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

Strings of the form $\underbrace{a \dots a}_n$

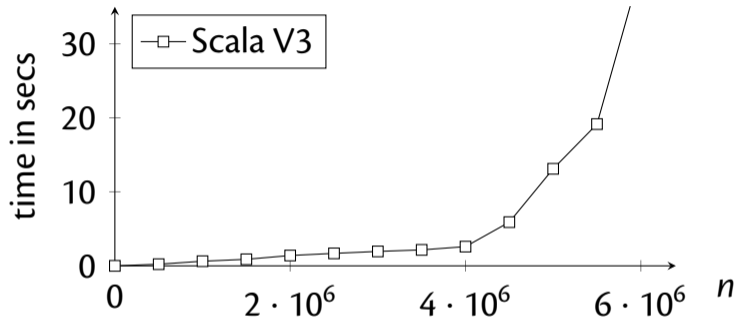
Same Example in Java 9+



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

Strings of the form $\underbrace{a \dots a}_n$

...and with Brzowski



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

Strings of the form $\underbrace{a \dots a}_n$

What is good about this Alg.

extends to most regular expressions, for example

$\sim r$ (next slide)

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...(another video)

Negation of Regular Expr's

$\sim r$ (everything that r cannot recognise)

$$L(\sim r) \stackrel{\text{def}}{=} UNIV - L(r)$$

$$nullable(\sim r) \stackrel{\text{def}}{=} \text{not}(nullable(r))$$

$$derc(\sim r) \stackrel{\text{def}}{=} \sim(derc r)$$

Negation of Regular Expr's

$\sim r$ (everything that r cannot recognise)

$$L(\sim r) \stackrel{\text{def}}{=} UNIV - L(r)$$

$$\text{nullable}(\sim r) \stackrel{\text{def}}{=} \text{not}(\text{nullable}(r))$$

$$\text{der } c(\sim r) \stackrel{\text{def}}{=} \sim(\text{der } c r)$$

Used often for recognising comments:

$$/ \cdot * \cdot (\sim ([a-z]^* \cdot * \cdot / \cdot [a-z]^*)) \cdot * \cdot /$$

The Specification for Matching

A regular expression r matches a string s provided:

$$s \in L(r)$$

matcher s r if and only if $s \in L(r)$

The Specification for Matching

A regular expression r matches a string s provided:

$$s \in L(r)$$

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

nullable and *der*

The central properties:

nullable(*r*) if and only if $\epsilon \in L(r)$

nullable and *der*

The central properties:

nullable(r) if and only if $\epsilon \in L(r)$

$$L(\text{der } cr) = \text{Derc}(L(r))$$

nullable and *der*

The central properties:

$\forall r. \text{ nullable}(r) \text{ if and only if } \epsilon \in L(r)$

$\forall r c. L(\text{der } c r) = \text{Derc}(L(r))$

Proofs about Rexprs

Remember their inductive definition:

$$r ::= \begin{array}{l} 0 \\ 1 \\ c \\ r_1 \cdot r_2 \\ r_1 + r_2 \\ r^* \end{array}$$

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

Proofs about Rexp (2)

P holds for 0 , 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

Assume $P(r)$ is the property:

$nullable(r)$ if and only if $[\] \in L(r)$

$$\text{der } c (r^*) \text{ def} = (\text{der } c r) \cdot (r^*)$$

Why in the example (slide 19) the first step is:

$$\text{der } a ((a \cdot b) + b)^* = (\text{der } a ((a \cdot b) + b)) \cdot r$$

and not

$$\text{der } a ((a \cdot b) + b) = (\text{der } a ((a \cdot b) + b)) \cdot (r^*)$$

Would it be possible to find and go over a few examples from the Brzozowski Algorithm, as it doesn't seem to be as simple as it sounds?

Is it possible to make a visual example of how using `simp()` function on a $(a^*)^*.b$ regular expression reduces its runtime? If not it's fine. I am just very surprised that it is so efficient.

Do you think the algorithm can be still improved (made faster)?

Do the regular expression matchers in Python and Java 8 have more features than the one implemented in this module? Or is there another reason for their inefficiency?

Will we discuss the broader Chomsky hierarchy of languages at some point?

