Lunch with a Lecturer (29 March)

I teach CFL (compilers) and PEP (Scala)

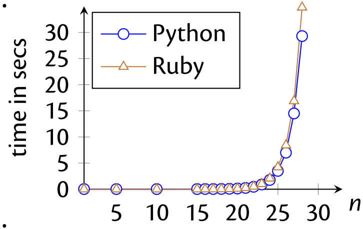
- did undergraduate in Germany
- Master in St Andrews
- PhD in Cambridge

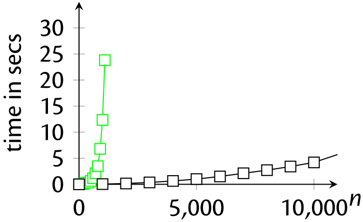
use mainly the Isabelle theorem prover in my work
(see 6CCS3VER)
write code in functional programming languages
(Scala, SML, Ocaml, Haskell)

Why Bother with Regexes?

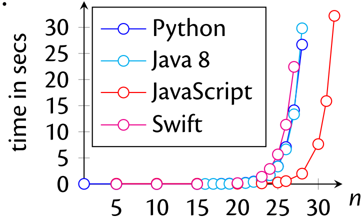
Ruby, Python, Java 8 | In PEP & CFL

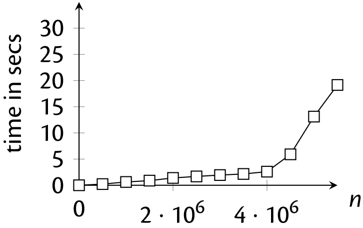
`[a?]{n}[a]{n}`:





`(a*)*b`:





matching with strings $\underbrace{\text{a...a}}_{n}$

Incidents

- a global outage on 2 July 2019 at **Cloudflare** (first one for six years)

```
(?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|true|false|
null|undefined|symbol|math)|\`|\-|\+)+[)]*;?((?:\s
|-|~|!|{}|\|\||\+)*.*(?:.*=.*)))
```

It serves more web traffic than Twitter, Amazon, Apple, Instagram, Bing & Wikipedia combined.

- on 20 July 2016 the **Stack Exchange** webpage went down because of an evil regular expression

"This conversation is interesting to me, and I've researched it a little bit... I also disagree with Dr. Urban on the cost/benefit of non-GC languages...[..]

But regardless, Scala is a lot slower than, say, C or Rust. To say it's not is basically wrong (imo)....[..] "

– Oliver Iliffe, discussion this year in PEP

"This conversation is interesting to me, and I've researched it a little bit... I also disagree with Dr. Urban on the cost/benefit of non-GC languages...[..]

But regardless, Scala is a lot slower than, say, C or Rust. To say it's not is basically wrong (imo)....[..] "

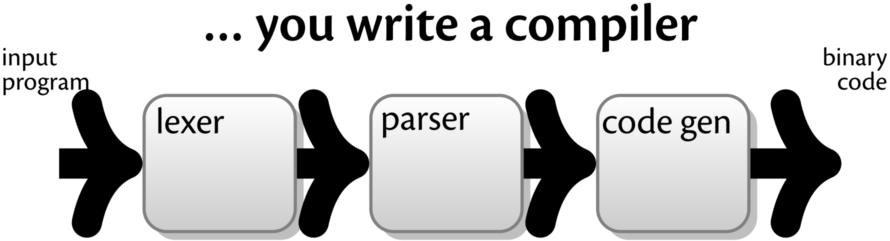
– Oliver Iliffe, discussion this year in PEP

Compilers and Formal Languages

Email:	christian.urban at kcl.ac.uk
Office Hour:	Fridays 11 – 12
Location:	N7.07 (North Wing, Bush House)
Slides & Progs:	KEATS
Pollev:	https://pollev.com/cfltutoratki576

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

The Goal of this Module...



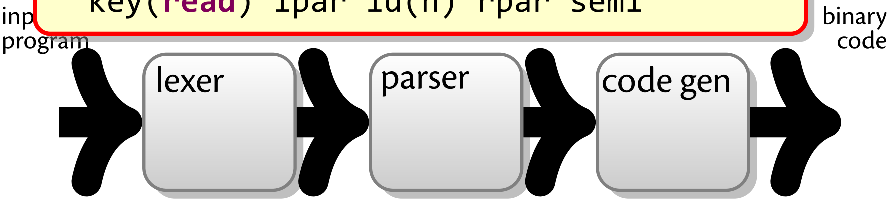
The Goal of this Module...

lexer input: a string

```
"read(n);"
```

lexer output: a sequence of tokens

```
key(read) lpar id(n) rpar semi
```



The Goal of this Module...

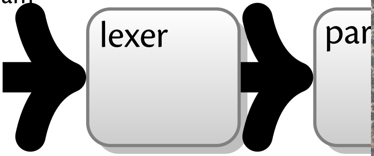
lexer input: a string

```
"read(n);"
```

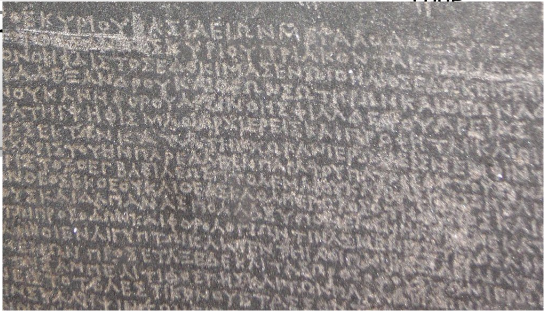
lexer output: a sequence of tokens

```
key(read) lpar id(n) rpar semi
```

inp
program



binary
code



lexing ⇒ recognising words (Stone of Rosetta)

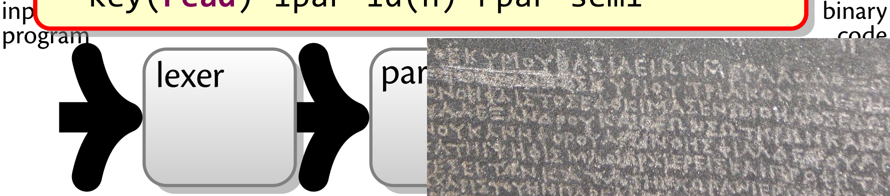
The Goal of this Module...

lexer input: a string

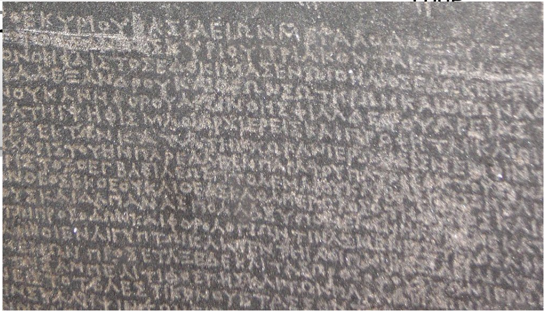
```
"read(n);"
```

lexer output: a sequence of tokens

```
key(read) lpar id(n) rpar semi
```



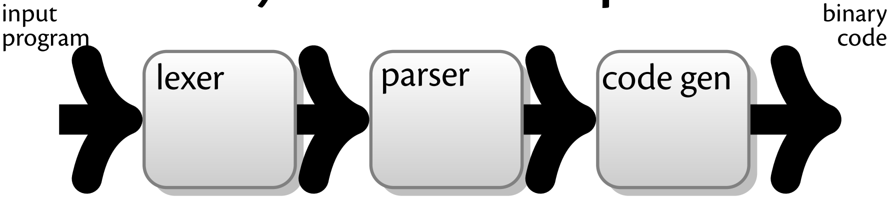
`if` ⇒ keyword
`iffoo` ⇒ identifier



lexing ⇒ recognising words (Stone of Rosetta)

The Goal of this Module...

... you write a compiler

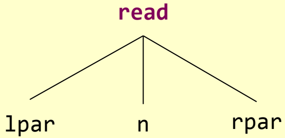


The Goal of this Module...

parser input: a sequence of tokens

```
key(read) lpar id(n) rpar semi
```

parser output: an abstract syntax tree

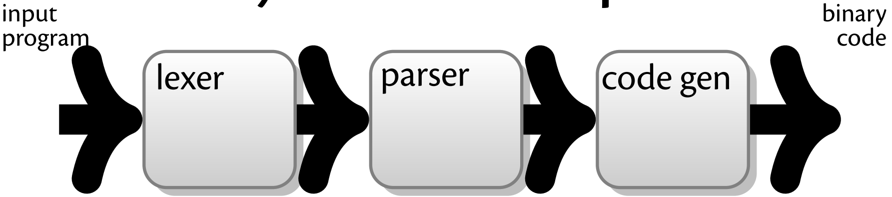


inp
pro

binary
code

The Goal of this Module...

... you write a compiler

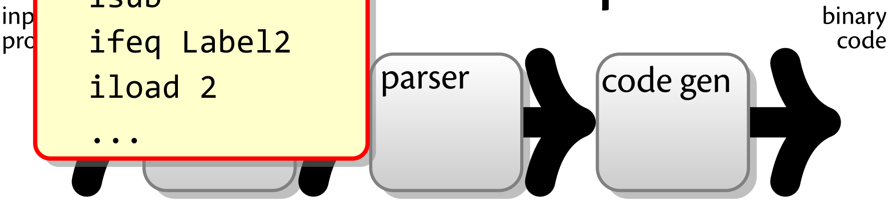


The Goal of this Module...

code generation:

```
istore 2
iload 2
ldc 10
isub
ifeq Label2
iload 2
...
```

...rite a compiler



The Goal of this Module...

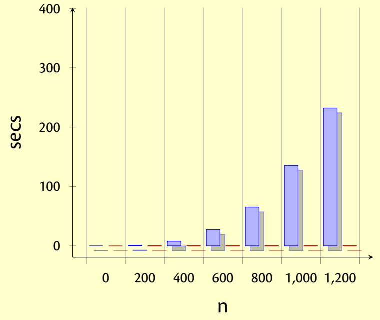
code generation:

```
istore 2
iload 2
ldc 10
isub
ifeq Label2
iload 2
...
```

write a compiler

inp
pro

parser



The Goal of this Module...

inp
prc

Compiler explorers, e.g.: `https://gcc.godbolt.org`

```
// Type your code here, or load an example.
int square(int num) {
    if (num % 2 == 0)
      { return num + num; }
    else
      { return num * num; }
}
```

```
square(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        and     eax, 1
        test    eax, eax
        jne     .L2
        mov     eax, DWORD PTR [rbp-4]
        add     eax, eax
        jmp     .L3
.L2:
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, eax
.L3:
        pop     rbp
        ret
```

source ⟶ binary

The Goal of this Module...

inp
pro

Compiler explorer for Java: `https://javap.yawk.at`

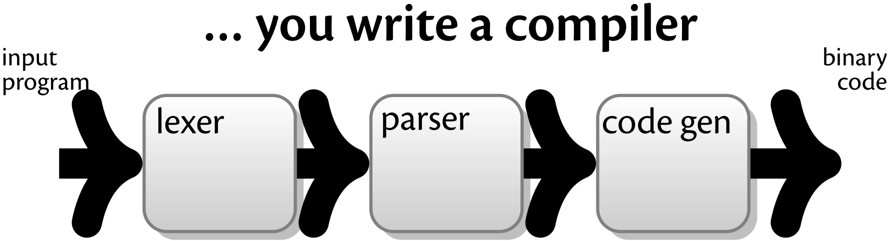
```java
import java.util.*;
import lombok.*;

public class Main {
    public Main() {
        int i = 0;
        i++;
    }
}
```

```
    Code:
      stack=1, locals=2, args_size=1
        start local 0 // Main this
         0: aload_0
         1: invokespecial #1
         4: iconst_0
         5: istore_1
        start local 1 // int i
         6: iinc          1, 1
         9: return
        end local 1 // int i
        end local 0 // Main this
```

source ⟶ byte code

The Goal of this Module...



Why Study Compilers?

John Regehr (Univ. Utah, LLVM compiler hacker) ☞

"...It's effectively a perpetual employment act for solid compiler hackers."

Why Study Compilers?

John Regehr (Univ. Utah, LLVM compiler hacker) 👉

> **"...It's effectively a perpetual employment act for solid compiler hackers."**

- **Hardware is getting weirder rather than getting clocked faster.**

  "Almost all processors are multicores nowadays and it looks like there is increasing asymmetry in resources across cores. Processors come with vector units, crypto accelerators etc. We have DSPs, GPUs, ARM big.little, and Xeon Phi. This is only scratching the surface."

Why Study Compilers?

John Regehr (Univ. Utah, LLVM compiler hacker) ☞

"...It's effectively a perpetual employment act for solid compiler hackers."

- **We're getting tired of low-level languages and their associated security disasters.**

  "We want to write new code, to whatever extent possible, in safer, higher-level languages. Compilers are caught right in the middle of these opposing trends: one of their main jobs is to help bridge the large and growing gap between increasingly high-level languages and increasingly wacky platforms."

"I enjoyed the module - it was genuinely the stand out academic experience of my undergraduate degree, and very much influenced my career interests. In fact I am currently working at ARM, in their Open Source Software group, on AArch64 specific optimisations for the Java/Kotlin compiler that forms part of the Android Runtime."

– Hari Limaye in year 2021/22

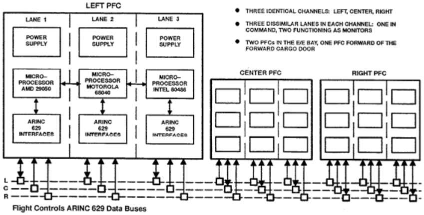
> *"I enjoyed the module - it was genuinely the stand out academic experience of my undergraduate degree, and very much influenced my career interests. In fact I am currently working at ARM, in their Open Source Software group, on AArch64 specific optimisations for the Java/Kotlin compiler that forms part of the Android Runtime."*
>
> *– Hari Limaye in year 2021/22*

Student numbers in CFL

2019:	32
2020:	59
2021:	109
2022:	121

Why Bother with Compilers?

Boeing 777's: First flight in 1994. They want to achieve triple redundancy for potential hardware faults. ☞



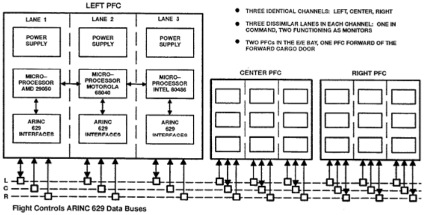
They compile 1 Ada program to

- Intel 80486
- Motorola 68040 (old Macintosh's)
- AMD 29050 (RISC chips used often in laser printers)

using 3 independent compilers.

Why Bother with Compilers?

Boeing 777's: First flight in 1994. They want to achieve triple redundancy for potential hardware faults. ☞



They compile 1 Ada program to

- Intel 80486
- Motorola 68040 (old Macintosh's)
- AMD 29050 (RISC chips used often in laser printers)

using 3 independent compilers.

Airbus uses C and static analysers. Recently started using CompCert.

What Do Compilers Do?

Remember BF*** from PEP?

`>`	$\Rightarrow$	move one cell right
`<`	$\Rightarrow$	move one cell left
`+`	$\Rightarrow$	increase cell by one
`-`	$\Rightarrow$	decrease cell by one
`.`	$\Rightarrow$	print current cell
`,`	$\Rightarrow$	input current cell
`[`	$\Rightarrow$	loop begin
`]`	$\Rightarrow$	loop end
	$\Rightarrow$	everything else is a comment

A “Compiler” for BF*** to C

`>`	$\Rightarrow$	`ptr++`
`<`	$\Rightarrow$	`ptr--`
`+`	$\Rightarrow$	`(*ptr)++`
`-`	$\Rightarrow$	`(*ptr)--`
`.`	$\Rightarrow$	`putchar(*ptr)`
`,`	$\Rightarrow$	`*ptr = getchar()`
`[`	$\Rightarrow$	`while(*ptr){`
`]`	$\Rightarrow$	`}`
	$\Rightarrow$	ignore everything else

```
char field[30000]
char *ptr = &field[15000]
```

Another "Compiler" for BF to C

`>…>`	⇒	`ptr += n`
`<…<`	⇒	`ptr -= n`
`+…+`	⇒	`(*ptr) += n`
`-…-`	⇒	`(*ptr) -= n`
`.`	⇒	`putchar(*ptr)`
`,`	⇒	`*ptr = getchar()`
`[`	⇒	`while(*ptr){`
`]`	⇒	`}`
	⇒	ignore everything else

```
char field[30000]
char *ptr = &field[15000]
```

A Brief Compiler History

- Turing Machines, 1936 (a tape as memory)
- Regular Expressions, 1956
- The first compiler for COBOL, 1957 (Grace Hopper)
- But surprisingly research papers are still published nowadays
- "Parsing: The Solved Problem That Isn't" ☞

Grace Hopper

(she made it to David Letterman's Tonight Show ☞)

Some Housekeeping

Exam will be online:

- final exam in January (35%)
- five CWs (65%)

Some Housekeeping

Exam will be online:

- final exam in January (35%)
- five CWs (65%)

Weekly Homework (optional):

- uploaded on KEATS, send answers via email, (try to!) respond individually
- all questions in the exam will be from the HWs!!

Homework

Last year(s): I did not give out solutions; students sent emails to me and I responded them individually

This year: We will do homework mainly during the Labs (TAs have the solutions)

Homework

Last year(s): I did not give out solutions; students sent emails to me and I responded them individually

This year: We will do homework mainly during the Labs (TAs have the solutions)

I will still choose the questions from the HW for the exam, but there might be some larger amount of deviation.

Some Housekeeping

Coursework (5 accounting for 65%):

- matcher (5%)
- lexer (10%)
- parser / interpreter (10%)
- JVM compiler (15%)
- LLVM compiler (25%)

Some Housekeeping

Coursework (5 accounting for 65%):

- matcher (5%)
- lexer (10%)
- parser / interpreter (10%)
- JVM compiler (15%)
- LLVM compiler (25%)

you can use any programming language you like (Haskell, Rust)

Some Housekeeping

Coursework (5 accounting for 65%):

- matcher (5%)
- lexer (10%)
- parser / interpreter (10%)
- JVM compiler (15%)
- LLVM compiler (25%)

you can use any programming language you like (Haskell, Rust)

you can use any code I show you and is uploaded to KEATS...**BUT NOTHING ELSE!**

Lectures 1 - 5

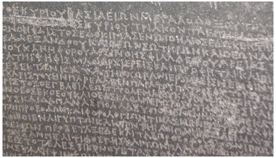
transforming strings into structured data

Lexing based on regular expressions

(recognising "words")

Parsing

(recognising "sentences")



Stone of Rosetta

Lectures 1 - 5

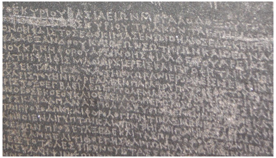
transforming strings into structured data

Lexing based on regular expressions

(recognising "words")

Parsing

(recognising "sentences")



Stone of Rosetta

Lectures 5 - 10

code generation for a small imperative and a small functional language

Interpreters

(directly runs a program)

Compilers

(generate JVM code and LLVM-IR code)

Familiar Regular Expresssions

```
[a-z0-9_\.-]+ @ [a-z0-9\.-]+ . [a-z\.]{2,6}
```

`re*`	matches 0 or more times
`re+`	matches 1 or more times
`re?`	matches 0 or 1 times
`re{n}`	matches exactly `n` number of times
`re{n,m}`	matches at least `n` and at most `m` times
`[...]`	matches any single character inside the brackets
`[^...]`	matches any single character not inside the brackets
`a-z A-Z`	character ranges
`\d`	matches digits; equivalent to `[0-9]`
`.`	matches every character except newline
`(re)`	groups regular expressions and remembers the matched text

Notation for REs

Some “innocent” examples

Let’s try two examples

`(a*)*b`   `[a?]{n}[a]{n}`

Some "innocent" examples

Let's try two examples

`(a*)*b`   `[a?]{n}[a]{n}`

and match them with strings of the form

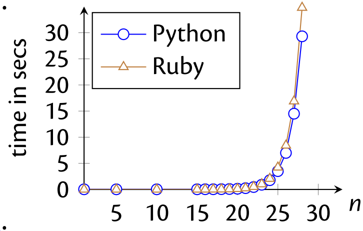
$$\texttt{a}, \texttt{aa}, \texttt{aaa}, \texttt{aaaa}, \texttt{aaaaa}, \underbrace{\texttt{a...a}}_{n}$$

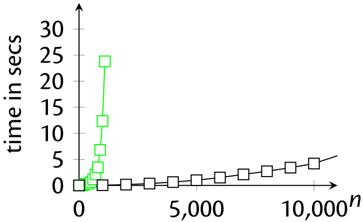
Why Bother with Regexes?

Ruby, Python, Java 8

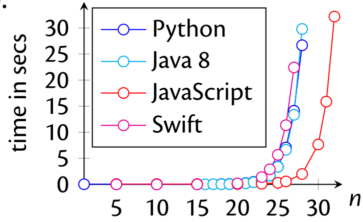
Us (after next lecture)

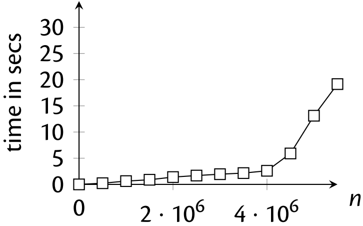
`[a?]{n}[a]{n}`:





`(a*)*b`:





matching with strings $\underbrace{\texttt{a...a}}_{n}$

Incidents

- a global outage on 2 July 2019 at **Cloudflare** (first one for six years)

```
(?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|true|false|
null|undefined|symbol|math)|\`|\-|\+)+[)]*;?((?:\s
|-|~|!|{}|\|\||\+)*.*(?:.*=.*)))
```

CLOUDFLARE®

It serves more web traffic than Twitter, Amazon, Apple, Instagram, Bing & Wikipedia combined. ☞

- on 20 July 2016 the **Stack Exchange** webpage went down because of an evil regular expression ☞

Evil Regular Expressions

- Regular expression Denial of Service (ReDoS)
- Some evil regular expressions:
  - `[a?]{n} [a]{n}`
  - `(a*)* b`
  - `([a-z]+)*`
  - `(a + aa)*`
  - `(a + a?)*`
- sometimes also called catastrophic backtracking
- this is a problem for Network Intrusion Detection systems, Cloudflare, StackExchange, Atom editor
- https://vimeo.com/112065252

(Basic) Regular Expressions

Their inductive definition:

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

(B

Their

```
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	character
	$\mid$	$r_1 + r_2$	alternative / choice
	$\mid$	$r_1 \cdot r_2$	sequence
	$\mid$	r^*	star (zero or more)

Strings

...are lists of characters. For example `"hello"`

$$[h, e, l, l, o] \text{ or just } hello$$

the empty string: $[]$ or `""`

the concatenation of two strings:

$$s_1 @ s_2$$

$foo @ bar = foobar$
$baz @ [] = baz$

Languages, Strings

- **Strings** are lists of characters, for example

  $[\,]$, abc (Pattern match: $c::s$)

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

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

- **Concatenation** of strings and languages

  $$foo \,@\, bar \quad = \quad foobar$$

  $$A \,@\, B \quad \stackrel{\text{def}}{=} \quad \{s_1 \,@\, s_2 \mid s_1 \in A \wedge s_2 \in B\}$$

Languages, Strings

- **Strings** are lists of characters, for example

  $[]$, abc (Pattern match: $c :: s$)

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

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

- **Concatenation** of strings and languages

  $$foo @ bar \quad = \quad foobar$$

  $$A @ B \stackrel{\text{def}}{=} \{s_1 @ s_2 \mid s_1 \in A \wedge s_2 \in B\}$$

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

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

Two Corner Cases

$$A\,@\,\{[]\} = \,?$$

Two Corner Cases

$$A \mathbin{@} \{[]\} = ?$$

$$A \mathbin{@} \{\} = ?$$

The Meaning of a Regex

...all the strings a regular expression can match.

$$
\begin{array}{rcl}
L(\mathbf{0}) & \stackrel{\text{def}}{=} & \{\} \\
L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{[]\} \\
L(c) & \stackrel{\text{def}}{=} & \{[c]\} \\
L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\
L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) \,@\, L(r_2) \\
L(r^*) & \stackrel{\text{def}}{=} &
\end{array}
$$

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

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

The Power Operation

- The ***nth* Power** of a language:

$$A^0 \stackrel{\text{def}}{=} \{[]\}$$
$$A^{n+1} \stackrel{\text{def}}{=} A \,@\, A^n$$

For example

$$A^4 = A \,@\, A \,@\, A \,@\, A \qquad (@\,\{[]\})$$
$$A^1 = A \qquad (@\,\{[]\})$$
$$A^0 = \{[]\}$$

The Meaning of a Regex

$$
\begin{aligned}
L(\mathbf{0}) &\stackrel{\text{def}}{=} \{\,\} \\
L(\mathbf{1}) &\stackrel{\text{def}}{=} \{[\,]\} \\
L(c) &\stackrel{\text{def}}{=} \{[c]\} \\
L(r_1 + r_2) &\stackrel{\text{def}}{=} L(r_1) \cup L(r_2) \\
L(r_1 \cdot r_2) &\stackrel{\text{def}}{=} \{s_1 @ s_2 \mid s_1 \in L(r_1) \wedge s_2 \in L(r_2)\} \\
L(r^*) &\stackrel{\text{def}}{=}
\end{aligned}
$$

The Meaning of a Regex

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

The Star Operation

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

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

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 Regex

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

The Meaning of Matching

A regular expression r matches a string s provided

$$s \in L(r)$$

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

Questions

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

  How many strings are in A^4 ?

Questions

- 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 ?

Questions

- Assume a set A contains 4 strings and a set B contains 7 strings. None of the strings is the empty string.
- How many strings are in $A\ @\ B$?

Questions?

TAs:	Huang Linh	(took the module last year)
	Alfredo Musumeci	
	Issa Kabir	

Coursework

Do we need to provide instructions on running the coursework files if we're using languages other than Scala? Thanks

Coursework

Do we need to provide instructions on running the coursework files if we're using languages other than Scala? Thanks

Zip-File for Coursework

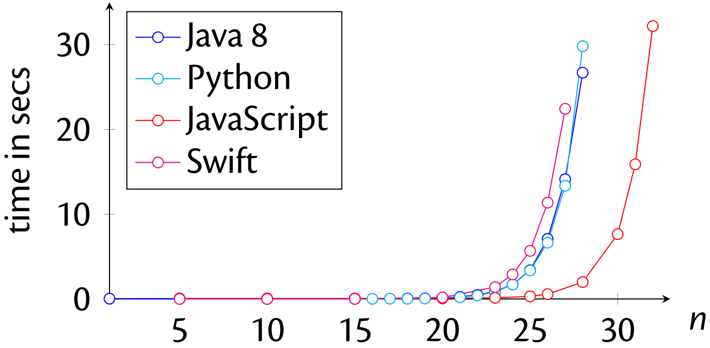
Please, please submit a zipfile that generates a subdirectory

`NameFamilyName`

What is the trick?

What was the trick to improve the evil regular expressions matcher to have such good results compared to other programming languages? Is it working better on casual regular expressions (the ones that Python and Java handle pretty well), too? Or was it just optimised for these evil ones?

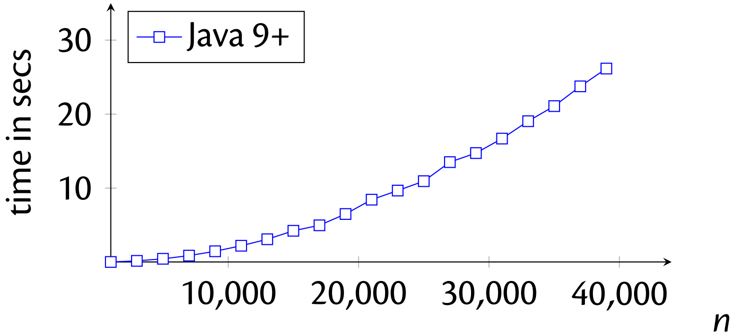
Thanks to Martin Mikusovic



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 $\underbrace{a \ldots a}_{n}$