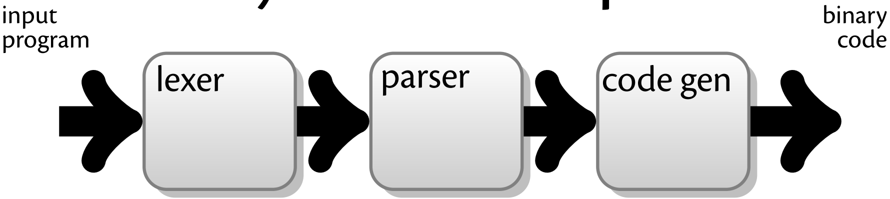
Compilers and Formal Languages

Email:	christian.urban at kcl.ac.uk
Slides & Progs:	KEATS

The Goal of this Module…

… you write a compiler



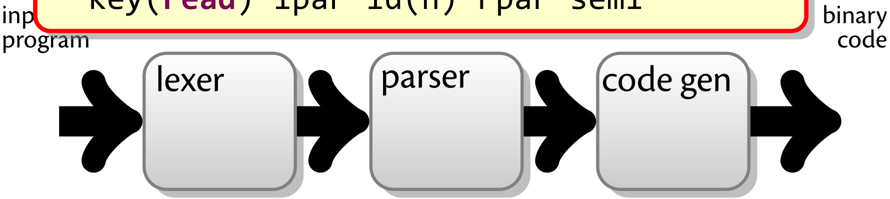
The Goal of this Module...

lexer input: a string

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

lexer output: a sequence of tokens

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



The Goal of this Module...

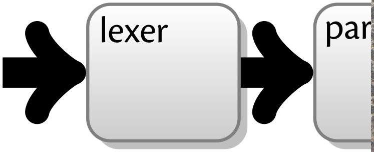
lexer input: a string

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

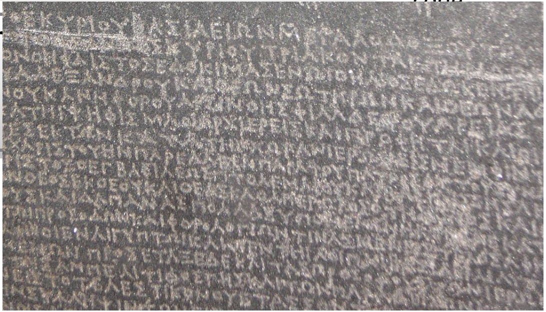
lexer output: a sequence of tokens

```
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
```

input program

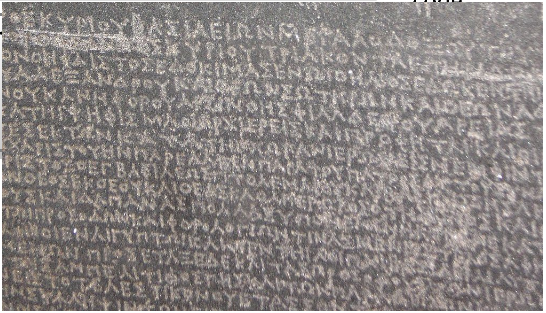
lexer

par

binary code

`if` ⇒ keyword

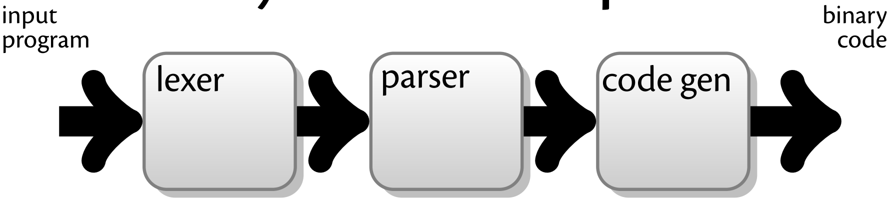
`iffoo` ⇒ identifier



lexing ⇒ recognising words (Stone of Rosetta)

The Goal of this Module...

... you write a compiler

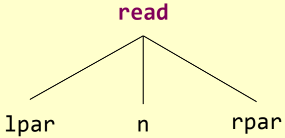


The Goal of this Module...

parser input: a sequence of tokens

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

parser output: an abstract syntax tree

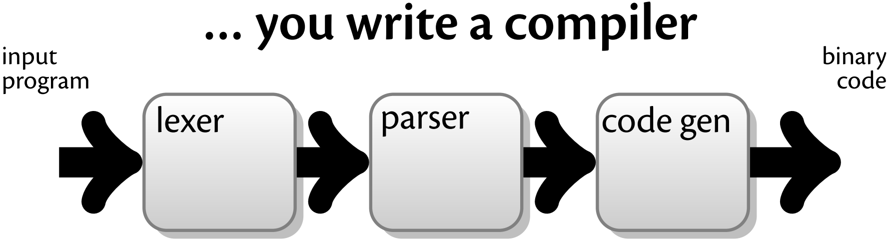


inp
pro

n

binary
code

The Goal of this Module...

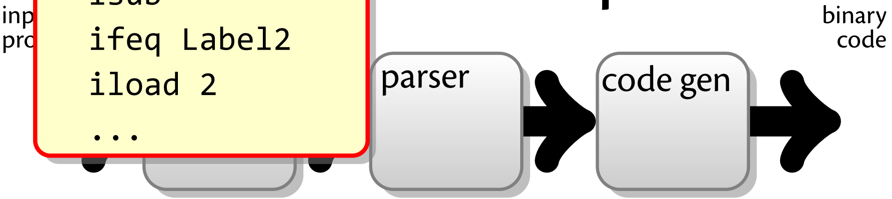


The Goal of this Module…

code generation:

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

rite a compiler



The Goal of this Module...

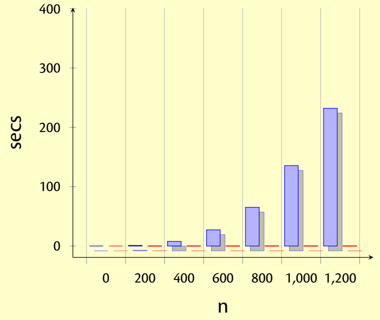
code generation:

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

rite a compiler

inp
pro

parser



The Goal of this Module...

inp
pro

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 $\longrightarrow$ **binary**

The Goal of this Module...

inp
pro

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

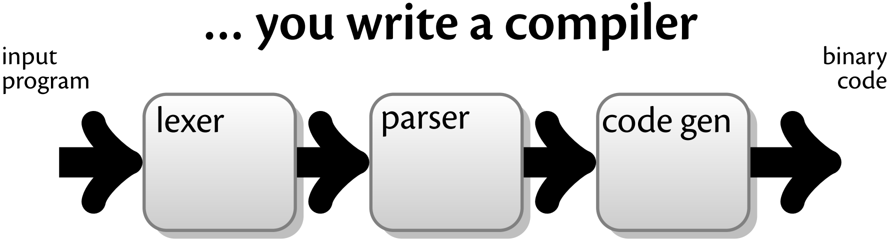
```java
import java.util.*;
import lombok.*;

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 $\longrightarrow$ 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) ☞

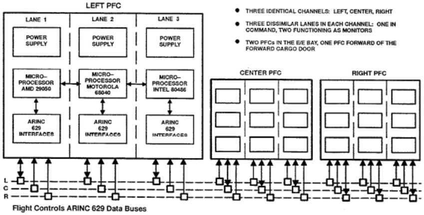
"...It's effectively a perpetual employment act for solid compiler hackers."

- **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."

Why Bother with Compilers?

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



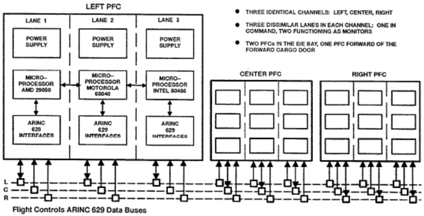
They compile 1 Ada program to

- 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

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

Exams will be online:

- final exam in January (30%)
- mid-term shortly after Reading Week (10%)

- weekly engagement (10%)

Some Housekeeping

Exams will be online:

- final exam in January (30%)
- mid-term shortly after Reading Week (10%)

- weekly engagement (10%)

Weekly Homework (optional):

- uploaded on KEATS, send answers via email, responded individually
- all questions in the exam and mid-term will be from the HW!!

Some Housekeeping

Coursework (5 accounting for 45%):

- matcher (5%)
- lexer (8%)
- parser / interpreter (10%)
- JVM compiler (10%)
- LLVM compiler (12%)

Some Housekeeping

Coursework (5 accounting for 45%):

- matcher (5%)
- lexer (8%)
- parser / interpreter (10%)
- JVM compiler (10%)
- LLVM compiler (12%)

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

Some Housekeeping

Coursework (5 accounting for 45%):

- matcher (5%)
- lexer (8%)
- parser / interpreter (10%)
- JVM compiler (10%)
- LLVM compiler (12%)

you can use any programming language you like (Haskell, Rust)
you can use any code I showed you and uploaded to KEATS...**BUT NOTHING ELSE!**

Some Housekeeping

Coursework (5 accounting for 45%):

- matcher (5%)
- lexer (8%)
- parser / interpreter (10%)
- JVM compiler (10%)
- LLVM compiler (12%)

you can use any programming language you like (Haskell, Rust)
you can use any code I showed you and uploaded to KEATS...**BUT NOTHING ELSE!**

Lectures 1 - 5

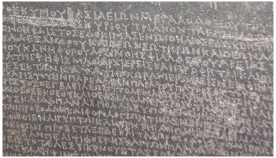
transforming strings into structured data

Lexing based on regular expressions

(recognising “words”)

Parsing

(recognising “sentences”)



Stone of Rosetta

Lectures 1 - 5

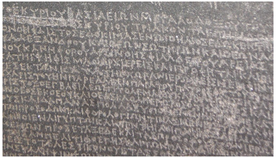
transforming strings into structured data

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

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

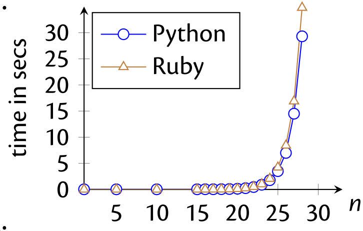
$$\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \mathtt{aaaaa}, \underbrace{\mathtt{a...a}}_{n}$$

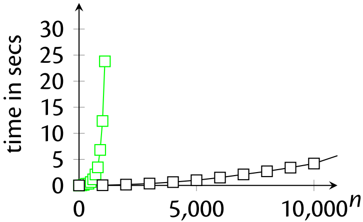
Why Bother with Regexes?

Ruby, Python, Java 8

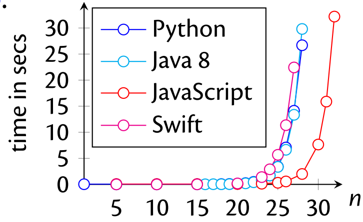
Us (after next lecture)

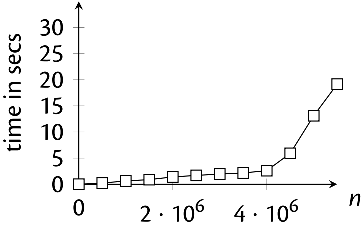
`[a?]{n}[a]{n}`:





`(a*)*b`:





matching with strings $\underbrace{a\ldots a}_{n}$

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

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

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

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

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

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

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

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

Two Corner Cases

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

Two Corner Cases

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

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

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:

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

For example

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

The Meaning of a Regex

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

The Meaning of a Regex

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

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

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

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?

TAs:	Anton Luca-Dorin	(took the module last year)
	Chengsong Tan	(PhD student working on derivatives)

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
```

Coursework

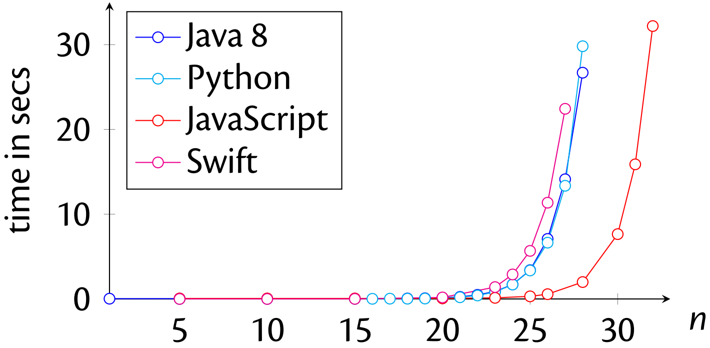
What is the purpose of the workshop session on the timetable?
Slightly confused about how to undertake cw1 and what exactly we should be implementing.
This is more for clarification of the cw1 structure, including the implementation and questions present in cw1.

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?

It was shown in the lectures that the pattern matching algorithms currently implemented in popular programming languages (Python, JS, Java, etc) are far slower than the algorithm we are going to be implementing in this module. My question is why do these programming languages not implement the algorithm that we are going to implement in this module?

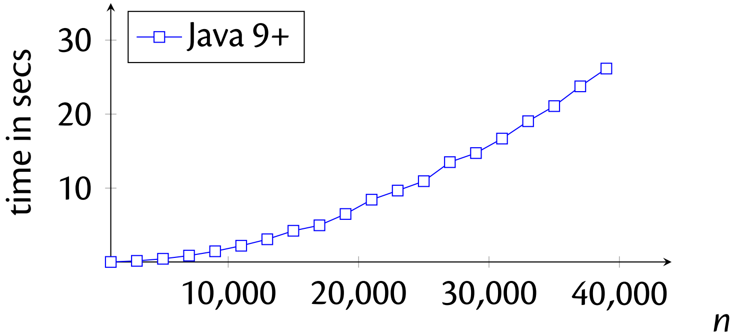
Thanks to Martin Mikusovic



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

Are there any (common) languages that have a built-in regex implementation matching the set of functions of a formal 'simple' regular expression, as opposed to an 'extended' regular expression implemented in most regex-supporting languages?

Passing Mark

I believe the assessment is 70% coursework (broken into 10% weekly stuff, 15% mid term exam and 45% CW in any programming language) and 30% January exam. However, I would like to know if we just need 40% overall to pass the module or pass the each component individually?

$\Rightarrow$ 40% overall

Regexes

Can we determine all the possible regular expressions matching a certain string? If we take into account all the possible ways to combine the operations: $\mathbf{0}$, $\mathbf{1}$, $r_1 + r_2$, $r_1 \cdot r_2$, r^*?

L + Equivalence

When we explain why two regular expressions are not equivalent, what method is better for us, using mathematics formulas or making an example?

Meaning of Regex and Operations

L

Can the function L be applied to anything other than regular expressions? For example would L(L(c)) return anything?

$\Rightarrow$ No

$(a?)\{n\} \cdot a\{n\}$

In the evil regexes section, is there any reason why in the regex `[a?]{n}[a]{n}` the square brackets are used? It is defined as a single character from the square brackets, however there is just one character, so it seems like it is not necessary. Maybe it is just necessary for the first part, because ? is a token instead of a character and we need to refer to a? as a “unit”? Could regular brackets be used instead? Is there any difference apart from the fact that it would create a group? Also, are the regexes `[a?]{n}` and `a{0,3}` equivalent?

Python + Parser Combinators (CW3)

Hi Christian,
I don't see a problem: you certainly have higher order functions and it is easy to implement algebraic data types using classes. As far as I can see that's all you need. You don't get the static types but that should be obvious. Basically if you can do it in LISP you can do it in Python. The only problem could be stack overflows due to a lack of tail recursion optimisation. On the other hand you can simulate laziness using generators.
Cheers, Thorsten

Trees `https://youtu.be/7tCNu4CnjVc`
Laziness `https://youtu.be/5jwV3zxXc8E`

What suggestions do you have for us to get the most out of this module, especially in the online format? I.e. form discussion groups, will you have office hours?

$\Rightarrow$ Discussion Forum on KEATS

online tutorial sessions

???

PL-groups for "exotic" langs

Where do most students struggle with this module? What will the format of the exam be? What is the most efficient way of studying for the exam? There are plenty of resources available on KEATS, but is there anything else you'd recommend us to study? Although (just by skimming the headings) the module seems to be a combination of practical and theoretical matters, exactly in what field would the syllabus be applied? Besides these questions and the ones other students asked, is there anything else we should know? Thank you!