Compilers and Formal Languages

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

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

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."

What are Compilers?

```
1  // Type your code here, or load an example.
2  int square(int num) {
3      if (num % 2 == 0)
4          { return num + num; }
5      else
6          { return num * num; }
7  }
```

```
1  square(int):
2          push    rbp
3          mov     rbp, rsp
4          mov     DWORD PTR [rbp-4], edi
5          mov     eax, DWORD PTR [rbp-4]
6          and     eax, 1
7          test    eax, eax
8          jne     .L2
9          mov     eax, DWORD PTR [rbp-4]
10         add     eax, eax
11         jmp     .L3
12 .L2:
13         mov     eax, DWORD PTR [rbp-4]
14         imul    eax, eax
15 .L3:
16         pop     rbp
17         ret
```

“source” ⟶ “binary”

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

Why Bother? Compilers & Boeings 777

First flight in 1994. They want to achieve triple redundancy in 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? Compilers & Boeings 777

First flight in 1994. They want to achieve triple redundancy in hardware faults.

They compile 1 Ada program to

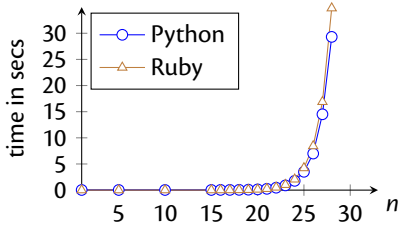
- Intel 80486
- Motorola 68040 (old Macintosh's)
- AMD 29050 (RISC chips used often in laser printers)

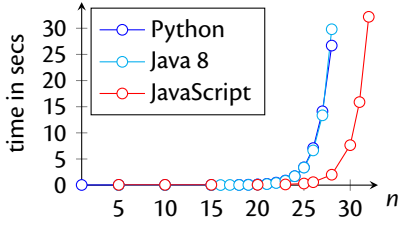
using 3 independent compilers.

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

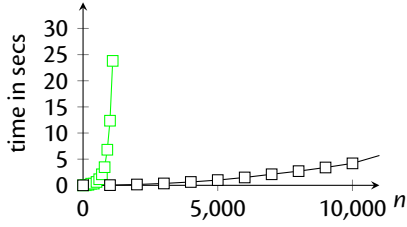
Why Bother?

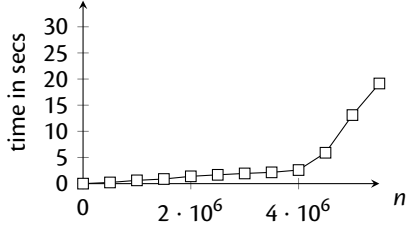
Ruby, Python, Java 8





Us (after next lecture)





matching `[a?]{n}[a]{n}` and `(a*)*b` against $\underbrace{\texttt{a...a}}_{n}$

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
|-|~|!|{}|\|\||\+)*.*(?:.*=.*)))
```

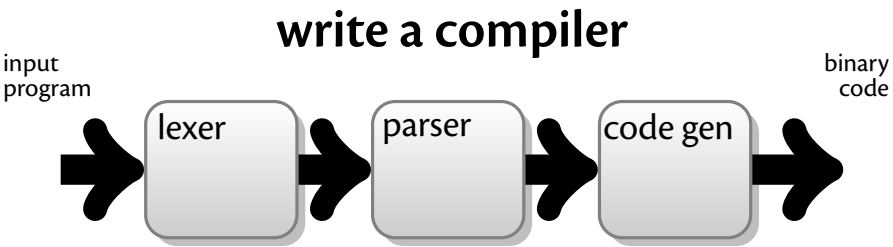
It serves more web traffic than Twitter, Amazon, Apple, Instagram, Bing & Wikipedia combined.

- 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)
- Evil regular expressions
  - $(a^{?\{n\}}) \cdot a^{\{n\}}$
  - $(a^*)^* \cdot b$
  - $([a-z]^+)^*$
  - $(a + a \cdot a)^*$
  - $(a + a^?)^*$
- sometimes also called catastrophic backtracking
- this is a problem for Network Intrusion Detection systems, Cloudflare, StackExchange, Atom editor
- `https://vimeo.com/112065252`

The Goal of this Module



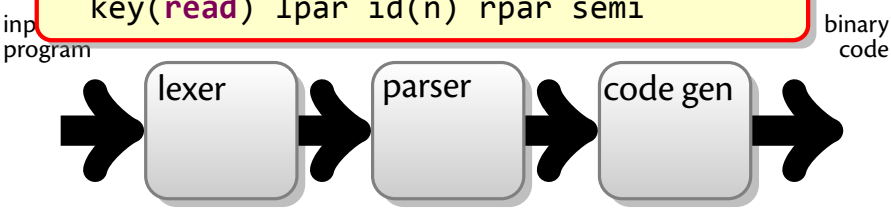
The Goal of this Module

lexer input: a string

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

lexer output: a sequence of tokens

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



The Goal of this Module

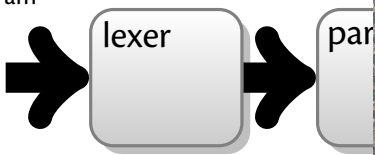
lexer input: a string

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

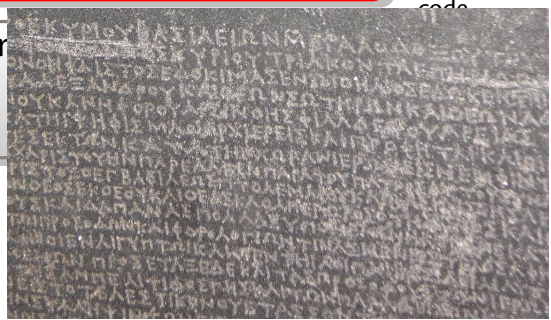
lexer output: a sequence of tokens

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

inp
program



binary
code



lexing ⇒ recognising words (Stone of Rosetta)

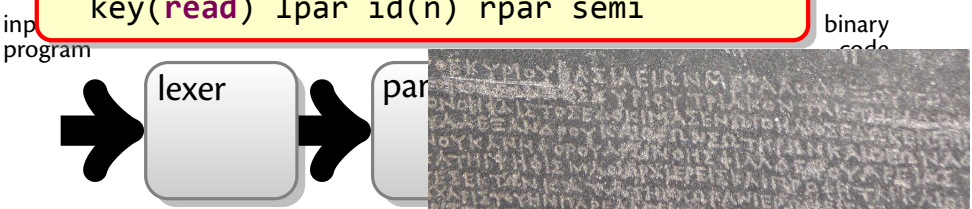
The Goal of this Module

lexer input: a string

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

lexer output: a sequence of tokens

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



`if` $\Rightarrow$ keyword
`iffoo` $\Rightarrow$ identifier

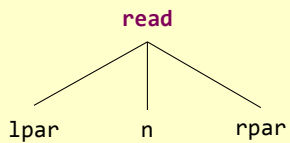
lexing $\Rightarrow$ recognising words (Stone of Rosetta)

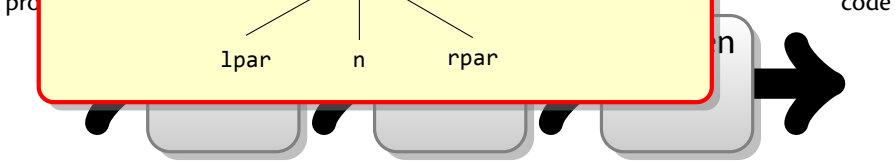
The Goal of this Module

parser input: a sequence of tokens

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

parser output: an abstract syntax tree



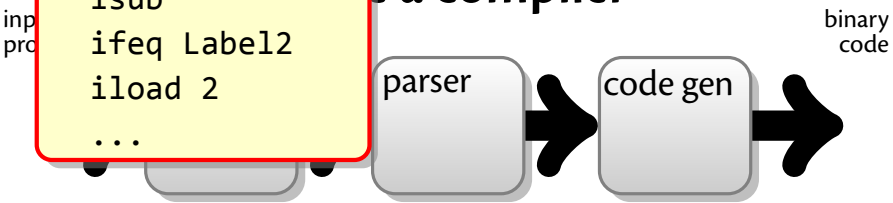


The Goal of this Module

code generator:

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

e a compiler



The Goal of this Module

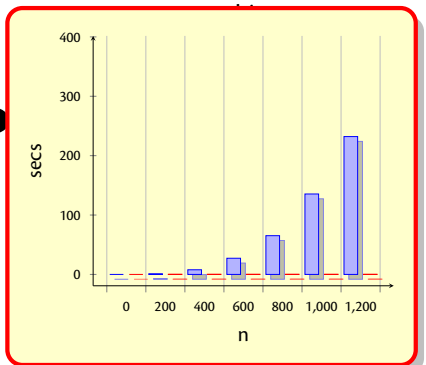
code generator:

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

e a compiler

inp
pro

parser



The Acad. Subject is Mature

- 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,

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***

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

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

Lectures 1 - 5

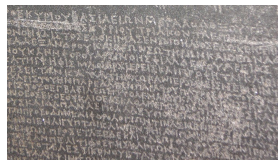
transforming strings into structured data

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 languages

Interpreters

(directly runs a program)

Compilers

(generates JVM code)

Familiar Regular Expr.

```
[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

A Regular Expression

- ... is a pattern or template for specifying strings

```
"https?://[^"]*"
```

matches for example

```
"http://www.foobar.com"
"https://www.tls.org"
```

but not

```
"http://www."foo"bar.com"
```

A Regular Expression

- ... is a pattern or template for specifying strings

```
"""https?://[^"]*""".r
```

matches for example

```
"http://www.foobar.com"
"https://www.tls.org"
```

but not

```
"http://www."foo"bar.com"
```

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)

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
```

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

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

  $$foo \mathbin{@} bar = foobar$$

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

The Meaning of a Regular Expression

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

The Meaning of a Regular Expression

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

The Meaning of a Regular Expression

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

(append on sets)

$$\{s_1 @ s_2 \mid s_1 \in L(r) \wedge s_2 \in L(r)^n\}$$

The Meaning of a Regular Expression

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

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

(append on sets)

$$\{s_1 @ s_2 \mid s_1 \in L(r) \wedge s_2 \in L(r)^n\}$$

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)

The Power Operation

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

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

For example

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

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 ?

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

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

Written Exam

- Accounts for 80%.
- The question "*Is this relevant for the exam*?" is very demotivating for the lecturer!
- Deal: Whatever is in the homework (and is not marked "*optional*") is relevant for the exam.
- Each lecture has also a handout. There are also handouts about notation and Scala.

Coursework

- Accounts for 20%. Two strands. Choose **one**!

Strand 1

- 4 programming tasks:
  - matcher (4%, 11.10.)
  - lexer (5%, 04.11.)
  - parser (5%, 22.11.)
  - compiler (6%, 13.12.)
- in any lang. you like, but I want to see the code

Strand 2

- one task: prove the correctness of a regular expression matcher in the Isabelle theorem prover
- 20%, submission on 13.12.

- Solving more than one strand will **not** give you more marks.

Lecture Capture

- Hope it works…

Lecture Capture

- Hope it works…actually no, it does not!

Lecture Capture

- Hope it works…actually no, it does not!
- It is important to use lecture capture wisely (it is only the "baseline"):
  - Lecture recordings are a study and revision aid.
  - Statistically, there is a clear and direct link between attendance and attainment: students who do not attend lectures, do less well in exams.
- Attending a lecture is more than watching it online – if you do not attend, you miss out!

Questions?