# Compilers and Formal Languages (1)

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

# 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?

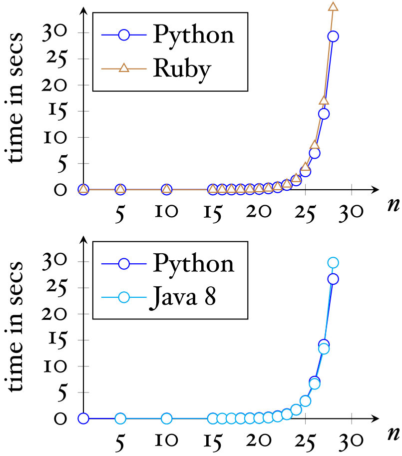
John Regehr (Univ. Utah, LLVM compiler hacker)

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

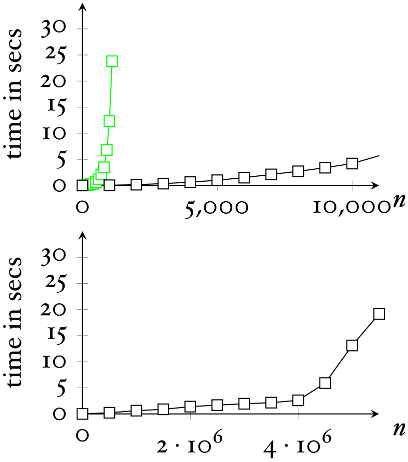
- **We're getting tired of low-level languages and their associated security disasters**
  - We want to write new code, to whatever extent possible, in safer, higher-level languages. Compilers are caught right in the middle of these opposing trends: one of their main jobs is to help bridge the large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

# Why Bother?

Ruby, Python, Java 8



Us (after next lecture)



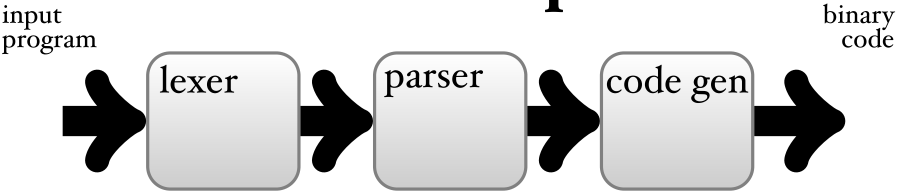
matching `[a?]{n}[a]{n}` and `(a*)*b` against $\underbrace{\texttt{a...a}}_{n}$

# Evil Regular Expressions

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

# The Goal of this Module

**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
```

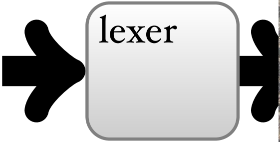
# The Goal of this Module

lexer input: a string

`"read(n);"`

lexer output: a sequence of tokens

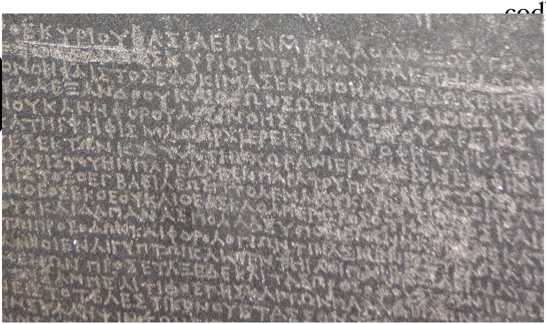
`key(read) lpar id(n) rpar semi`



input program

lexer

binary code



lexing ⇒ recognising words (Stone of Rosetta)

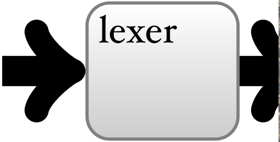
# The Goal of this Module

lexer input: a string

`"read(n);"`

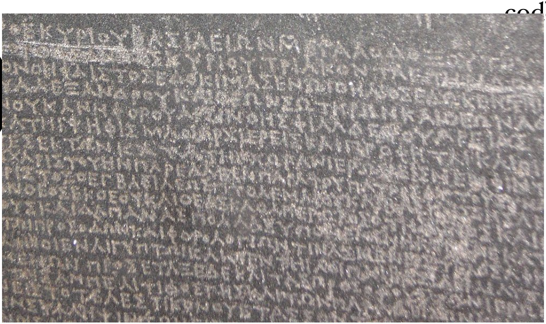
lexer output: a sequence of tokens

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



`if` ⇒ keyword

`iffoo` ⇒ identifier



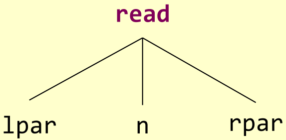
lexing ⇒ recognising words (Stone of Rosetta)

# The Goal of this Module

parser input: a sequence of tokens

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

parser output: an abstract syntax tree

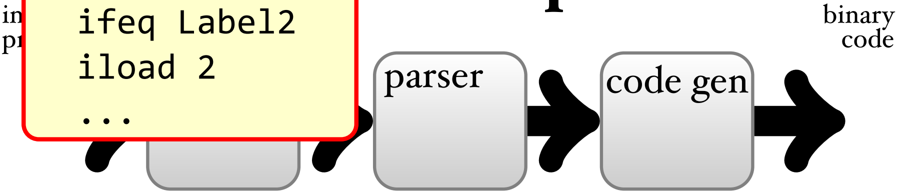


# The Goal of this Module

code generator:

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

## e a compiler



# The Goal of this Module

code generator:

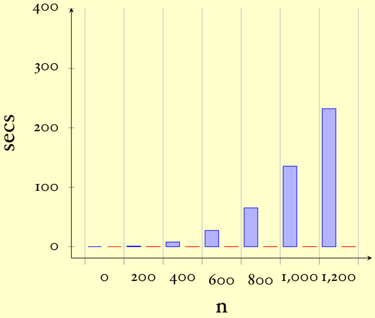
```
istore 2
iload 2
ldc 10
isub
ifeq Label2
iload 2
...
```

## e a compiler

in
pr

parse

y
e



# The Acad. Subject is Mature

- Turing Machines, 1936
- 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,

# Lectures 1 - 5

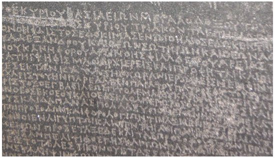
transforming strings into structured data

**Lexing** based on regular expressions

(recognising "words")

**Parsing**

(recognising "sentences")



Stone of Rosetta

# 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 |

# Today

- While the ultimate goal is to implement a small compiler for the JVM …

Let's start with:

- a web-crawler
- an email harvester

# A Web-Crawler

1. given an URL, read the corresponding webpage
2. extract all links from it
3. call the web-crawler again for all these links

# A Web-Crawler

1. given an URL, read the corresponding webpage
2. if not possible print, out a problem
3. if possible, extract all links from it
4. call the web-crawler again for all these links

# A Web-Crawler

1. given an URL, read the corresponding webpage
2. if not possible print, out a problem
3. if possible, extract all links from it
4. call the web-crawler again for all these links

(we need a bound for the number of recursive calls)
(the purpose is to check all links on my own webpage)

Server

GET request

webpage

POST data

Browser

# Scala

A simple Scala function for reading webpages:

```
import io.Source

def get_page(url: String) : String = {
  Source.fromURL(url)(”ISO-8859-1”).take(10000).mkString
}
```

# Scala

A simple Scala function for reading webpages:

```
import io.Source

def get_page(url: String) : String = {
  Source.fromURL(url)(”ISO-8859-1”).take(10000).mkString
}

get_page(”””https://nms.kcl.ac.uk/christian.urban/”””)
```

# Scala

A simple Scala function for reading webpages:

```
import io.Source

def get_page(url: String) : String = {
  Source.fromURL(url)(”ISO-8859-1”).take(10000).mkString
}

get_page(”””https://nms.kcl.ac.uk/christian.urban/”””)
```

A slightly more complicated version for handling errors:

```
def get_page(url: String) : String = {
 Try(Source.fromURL(url)(”ISO-8859-1”).take(10000).mkString).
    getOrElse { println(s”  Problem with: $url”); ””}
}
```

# 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”`

# Finding Operations in Scala

`rexp.findAllIn(string)`

returns a list of all (sub)strings that match the regular expression

`rexp.findFirstIn(string)`

returns either

- `None` if no (sub)string matches or
- `Some(s)` with the first (sub)string

```
val http_pattern = ””””https?://[^”]*””””.r

def unquote(s: String) = s.drop(1).dropRight(1)

def get_all_URLs(page: String) : Set[String] =
  http_pattern.findAllIn(page).map(unquote).toSet

def crawl(url: String, n: Int) : Unit = {
  if (n == 0) ()
  else {
    println(s”Visiting: $n $url”)
    for (u <- get_all_URLs(get_page(url))) crawl(u, n - 1)
  }
}

crawl(some_start_URL, 2)
```

A version that only crawls links in “my” domain:

```
val my_urls = ”””urban”””.r

def crawl(url: String, n: Int) : Unit = {
  if (n == 0) ()
  else if (my_urls.findFirstIn(url) == None) {
    println(s”Visiting: $n $url”)
    get_page(url); ()
  }
  else {
    println(s”Visiting: $n $url”)
    for (u <- get_all_URLs(get_page(url))) crawl(u, n - 1)
  }
}
```

## A little email harvester:

```
val http_pattern = ””””https?://[^”]*””””.r
val email_pattern =
  ”””([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})”””.r

def print_str(s: String) =
  if (s == ””) () else println(s)

def crawl(url: String, n: Int) : Unit = {
  if (n == 0) ()
  else {
    println(s”Visiting: $n $url”)
    val page = get_page(url)
    print_str(email_pattern.findAllIn(page).mkString(”\n”))
    for (u <- get_all_URLs(page).par) crawl(u, n - 1)
  }
}
```

http://net.tutsplus.com/tutorials/other/8-regular-expressions-you-should-know/

# 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) |

Th

```
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

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

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

# The Meaning of a Regular Expression

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

# The Meaning of a Regular Expression

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

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

# The Meaning of a Regular Expression

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

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

(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)

# 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

- four programming tasks:
  - matcher (4%, 12.10.)
  - lexer (5%, 02.11.)
  - parser (5%, 23.11.)
  - compiler (6%, 14.12.)
- 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 14.12.

- 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?