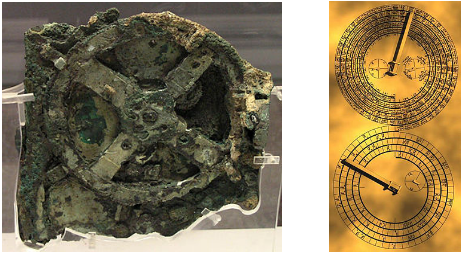
# Automata and Formal Languages (1)



Antikythera automaton, 100 BC (Archimedes?)


| | |
|---|---|
| Email: | christian.urban at kcl.ac.uk |
| Office: | S1.27 (1st floor Strand Building) |
| Slides: | KEATS |

# The Goal of this Course

## A Compiler

# The Goal of this Course

lexer input: a string

”read(n);”

lexer output: a sequence of tokens

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

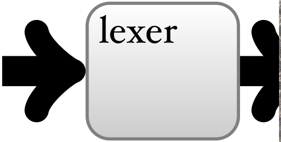
# The Goal of this Course

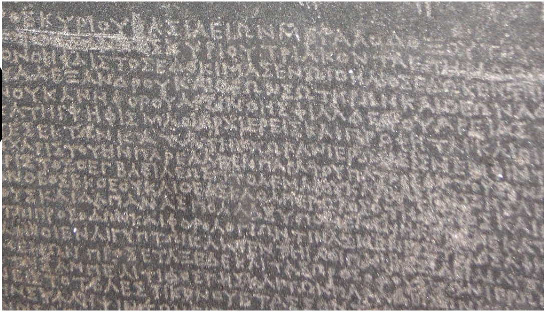
lexer input: a string

”read(n);”

lexer output: a sequence of tokens

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



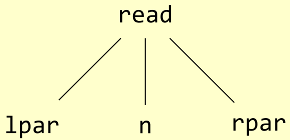



lexing ⇒ recognising words (Stone of Rosetta)

# The Goal of this Course

parser input: a sequence of token

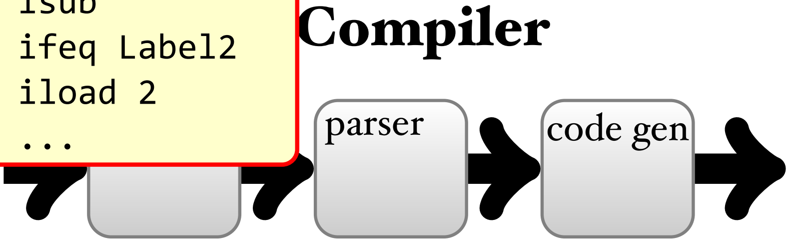
parser output: an abstract syntax tree




# The Goal of this Course

code generator:

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



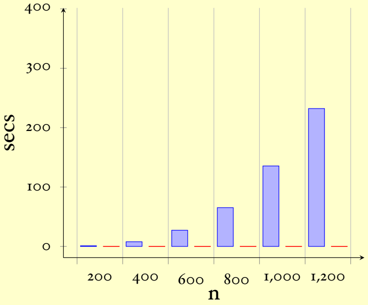
# The Goal of this Course

code generator:

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

**Compiler**

parse



# The subject is quite old

- Turing Machines, 1936
- Regular Expressions, 1956
- The first compiler for COBOL, 1957 (Grace Hopper)
- But surprisingly research papers are still published nowadays

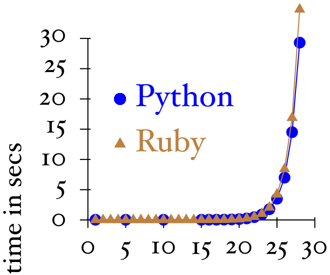
Grace Hopper

(she made it to David Letterman's Tonight Show,

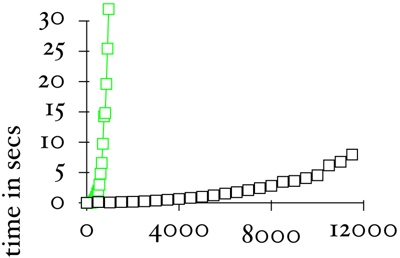
http://www.youtube.com/watch?v=aZOxtURhfEU)

# Why Bother?

Ruby, Python
and Others




Us (after this course)




matching `[a?]{n}[a]{n}` against $\underbrace{\texttt{a...a}}_{n}$

# Lectures 1 - 5

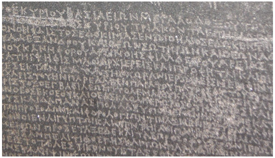
transforming strings into structured data

**Lexing**

(recognising "words")

**Parsing**

(recognising "sentences")



Stone of Rosetta

# Lectures 1 - 5

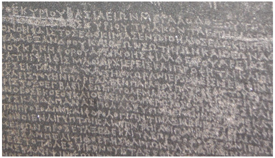
transforming strings into structured data

**Lexing** baseed 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-zA-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

- the ultimate goal is to implement a small compiler (a really small one for the JVM)

Let's start with:

- a web-crawler
- an email harvester
- a web-scraper

# 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

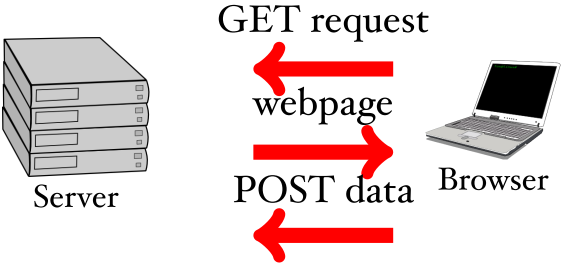
# A Web-Crawler

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)


GET request
webpage
POST data
Server
Browser

# Scala

A simple Scala function for reading webpages:

```
import io.Source

def get_page(url: String) : String = {
  Source.fromURL(url).take(10000).mkString
}
```

# Scala

A simple Scala function for reading webpages:

```
import io.Source

def get_page(url: String) : String = {
  Source.fromURL(url).take(10000).mkString
}

get_page("""http://www.inf.kcl.ac.uk/staff/urbanc/""")
```

# Scala

A simple Scala function for reading webpages:

```
import io.Source

def get_page(url: String) : String = {
  Source.fromURL(url).take(10000).mkString
}

get_page(""""http://www.inf.kcl.ac.uk/staff/urbanc/"""")
```

A slightly more complicated version for handling errors properly:

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

# Why Scala?

twitter
LinkedIn
theguardian
Morgan Stanley
CREDIT SUISSE
EDF ENERGY
Novell
foursquare
HSBC

...

# Why Scala?

twitter
Linked in
theguardian
Morgan Stanley
CREDIT SUISSE
EDF ENERGY
Novell
foursquare
HSBC

...

# Why Scala?

twitter
LinkedIn
theguardian
Morgan Stanley
CREDIT SUISSE
EDF ENERGY
Novell
foursquare
HSBC

...


5 yrs {
2013: 1%
2014: 3%
2015: 9%
2016: 27%
2017: 81%
2018: 243% 😁

# Why Scala?

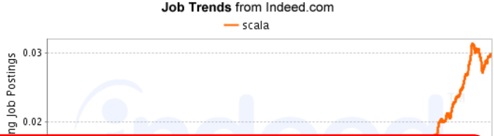
twitter
LinkedIn
theguardian
Morgan Stanley
CREDIT SUISSE
EDF ENERGY
Novell
foursquare
HSBC
...


5 yrs {
2013: 1%
2014: 3%
2015: 9%
2016: 27%
2017: 81%
}
2018: 243%

**in London today:** 1 Scala job for every 30 Java jobs;
Scala programmers seem to get up to 20% better salary

# Why Scala?

twitter
LinkedIn
theguardian
Morgan
Credit S
EDF ENERGY
Nove
foursquare
HSBC
...




Scala is a functional and object-oriented programming language; compiles to the JVM; does not need null-pointer exceptions; a course on Coursera
http://www.scala-lang.org

2016: 27%
2017: 81%
2018: 243% 😁

**in London today:** 1 Scala job for every 30 Java jobs;
Scala programmers seem to get up to 20% better salary

# A Regular Expression

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

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

matches for example

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

# A Regular Expression

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

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

matches for example

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

# Finding Operations

`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

```scala
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 = """urbanc""".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 my_urls = ”””urbanc”””.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:

$$
\begin{array}{rcll}
r & ::= & \varnothing & \text{null} \\
  & \mid & \epsilon & \text{empty string / "" / [ ]} \\
  & \mid & c & \text{character} \\
  & \mid & r_1 \cdot r_2 & \text{sequence} \\
  & \mid & r_1 + r_2 & \text{alternative / choice} \\
  & \mid & r^* & \text{star (zero or more)}
\end{array}
$$

# Regular Expressions

In Scala:

```
def OPT(r: Rexp) = ALT(r, EMPTY)

def NTIMES(r: Rexp, n: Int) : Rexp = n match {
  case 0 => EMPTY
  case 1 => r
  case n => SEQ(r, NTIMES(r, n - 1))
}
```

# Strings

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

$$[h, e, l, l, o]$$

the empty string: $[]$ or ""

the concatenation of two strings:

$$s_1 @ s_2$$

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

# The Meaning of a Regular Expression

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

# The Meaning of a Regular Expression

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

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

# The Meaning of a Regular Expression

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

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

(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

$$L(\varnothing) \stackrel{\text{def}}{=} \varnothing$$
$$L(\epsilon) \stackrel{\text{def}}{=} \{[]\}$$
$$L(c) \stackrel{\text{def}}{=} \{[c]\}$$
$$L(r_1 + r_2) \stackrel{\text{def}}{=} L(r_1) \cup L(r_2)$$
$$L(r_1 \cdot r_2) \stackrel{\text{def}}{=} \{s_1 @ s_2 \mid s_1 \in L(r_1) \wedge s_2 \in L(r_2)\}$$
$$L(r^*) \stackrel{\text{def}}{=} \bigcup_{n \geq 0} L(r)^n$$

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

(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

$$s \in L(r)$$

# Written Exam

- Accounts for 75%.
- You will understand 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.

# Coursework

- Accounts for 25%. Two strands. Choose **one**!

**Strand 1**

- four programming subtasks:
  - matcher (5%, 16.10.)
  - lexer (5%, 06.11.)
  - parser (5%, 27.11.)
  - compiler (10%, 11.12.)

**Strand 2**

- one task: prove the correctness of a regular expression matcher in the Isabelle theorem prover
- 25%, submission 11.12.

- Solving more than one strand will **not** give you more marks.
- The exam will contain in much, much smaller form elements from both (but will also be in lectures and HW).

# Questions?