Compilers and Formal Languages

Email:	christian.urban at kcl.ac.uk
Office Hour:	Fridays 12 – 14
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

While Tokens

$$\texttt{WHILE\_REGS} \stackrel{\text{def}}{=} (\texttt{("k" : KEYWORD)} + \texttt{("i" : ID)} + \texttt{("o" : OP)} + \texttt{("n" : NUM)} + \texttt{("s" : SEMI)} + \texttt{("p" : (LPAREN + RPAREN))} + \texttt{("b" : (BEGIN + END))} + \texttt{("w" : WHITESPACE)})^*$$

The Goal of this Course

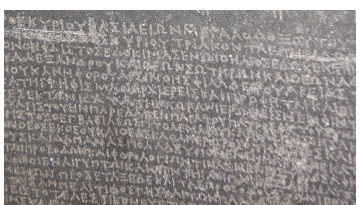
Write a compiler

Today a lexer.

The Goal of this Course

Write a compiler

Today a lexer.



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

Regular Expressions

In programming languages they are often used to recognise:

- operands, digits
- identifiers
- numbers (non-leading zeros)
- keywords
- comments

`http://www.regexper.com`

Lexing: Test Case

```
write "Fib";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

"if true then then 42 else +"

KEYWORD:

  if, then, else,

WHITESPACE:

  " ", \n,

IDENTIFIER:

  $LETTER \cdot (LETTER + DIGIT + \_)^*$

NUM:

  $(NONZERODIGIT \cdot DIGIT^*) + 0$

OP:

  +, -, *, %, <, <=

COMMENT:

  $/* \cdot \sim(ALL^* \cdot (*/) \cdot ALL^*) \cdot */$

"if true then then 42 else +"

```
KEYWORD(if),
WHITESPACE,
IDENT(true),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
NUM(42),
WHITESPACE,
KEYWORD(else),
WHITESPACE,
OP(+)
```

"if true then then 42 else +"

```
KEYWORD(if),
IDENT(true),
KEYWORD(then),
KEYWORD(then),
NUM(42),
KEYWORD(else),
OP(+)
```

There is one small problem with the tokenizer. How should we tokenize…?

```
"x-3"
```

```
ID: …
OP:
  "+", "-"
NUM:
  (NONZERODIGIT · DIGIT*) + "0"
NUMBER:
  NUM + ("-" · NUM)
```

The same problem with

$$(ab + a) \cdot (c + bc)$$

and the string abc.

The same problem with

$$(ab + a) \cdot (c + bc)$$

and the string *abc*.

Or, keywords are `if` etc and identifiers are letters followed by "letters + numbers + _"*

```
if      iffoo
```

POSIX: Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as the next token.

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

POSIX: Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as the next token.

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

most posix matchers are buggy

http://www.haskell.org/haskellwiki/Regex_Posix

POSIX: Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as the next token.

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

most posix matchers are buggy

http://www.haskell.org/haskellwiki/Regex_Posix

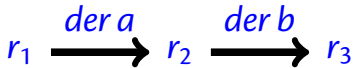
traditional lexers are fast, but hairy

Sulzmann & Lu Lexer

We want to match the string abc using r_1:

Sulzmann & Lu Lexer

We want to match the string abc using r_1:

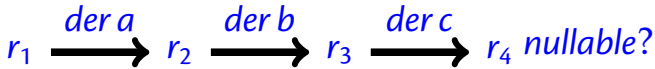


Sulzmann & Lu Lexer

We want to match the string abc using r_1:

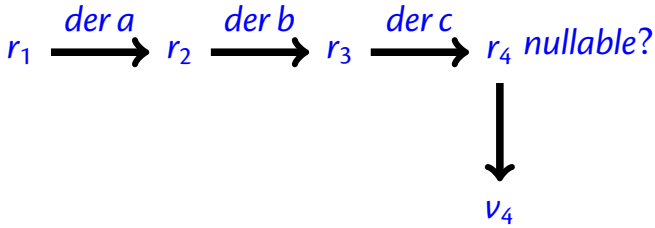
Sulzmann & Lu Lexer

We want to match the string *abc* using r_1:



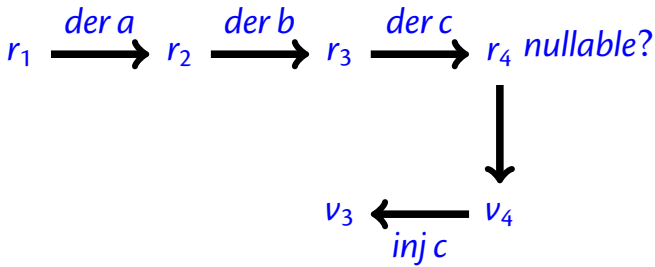
Sulzmann & Lu Lexer

We want to match the string abc using r_1:



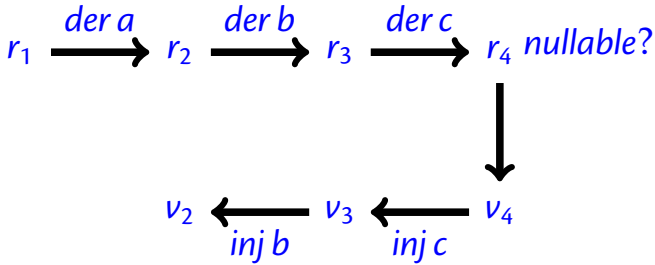
Sulzmann & Lu Lexer

We want to match the string *abc* using r_1:



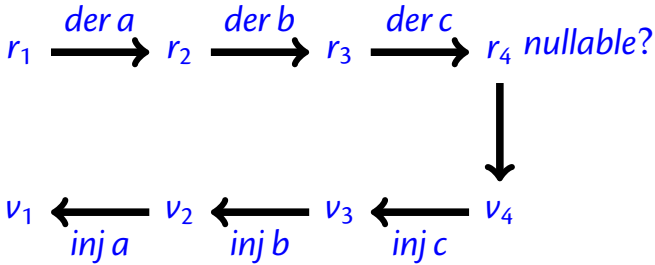
Sulzmann & Lu Lexer

We want to match the string *abc* using r_1:



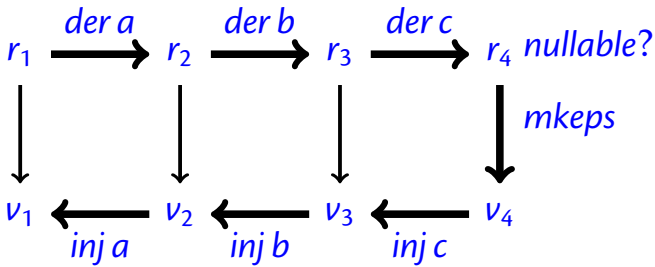
Sulzmann & Lu Lexer

We want to match the string *abc* using r_1:



Sulzmann & Lu Lexer

We want to match the string abc using r_1:



Regexes and Values

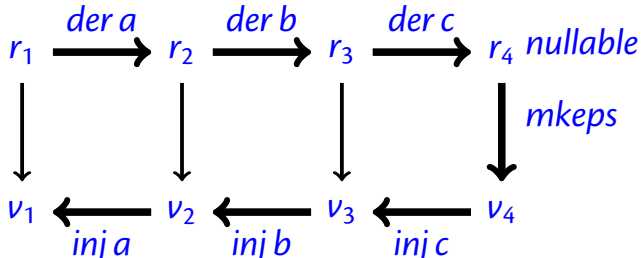
Regular expressions and their corresponding values:

$$
\begin{array}{rcl}
r & ::= & \mathbf{0} \\
 & | & \mathbf{1} \\
 & | & c \\
 & | & r_1 \cdot r_2 \\
 & | & r_1 + r_2 \\
 & | & r^* \\
 & | & (r)^{\{n\}}
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
 & & \mathit{Empty} \\
 & | & \mathit{Char}(c) \\
 & | & \mathit{Seq}(v_1, v_2) \\
 & | & \mathit{Left}(v) \\
 & | & \mathit{Right}(v) \\
 & | & \mathit{Stars}\,[] \\
 & | & \mathit{Stars}\,[v_1, \ldots v_n] \\
 & | & \mathit{Rec}(x, v)
\end{array}
$$

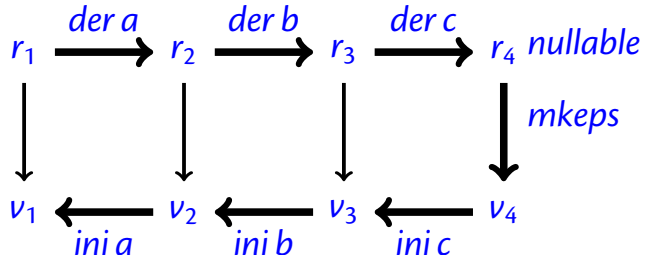
```
abstract class Rexp
case object ZERO extends Rexp
case object ONE extends Rexp
case class CHAR(c: Char) extends Rexp
case class ALT(r1: Rexp, r2: Rexp) extends Rexp
case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
case class STAR(r: Rexp) extends Rexp

abstract class Val
case object Empty extends Val
case class Chr(c: Char) extends Val
case class Sequ(v1: Val, v2: Val) extends Val
case class Left(v: Val) extends Val
case class Right(v: Val) extends Val
case class Stars(vs: List[Val]) extends Val
```

r_1: $a \cdot (b \cdot c)$
r_2: $\mathbf{1} \cdot (b \cdot c)$
r_3: $(\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$
r_4: $(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$



r_1: $a \cdot (b \cdot c)$
r_2: $\mathbf{1} \cdot (b \cdot c)$
r_3: $(\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$
r_4: $(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$



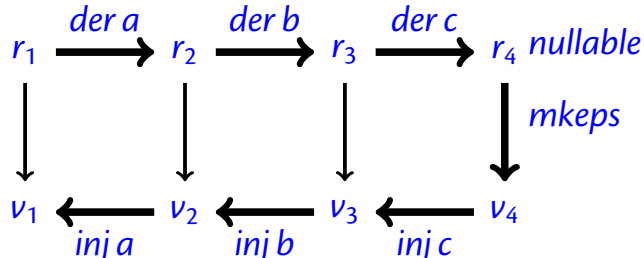
v_1: $Seq(Char(a), Seq(Char(b), Char(c)))$
v_2: $Seq(Empty, Seq(Char(b), Char(c)))$
v_3: $Right(Seq(Empty, Char(c)))$
v_4: $Right(Right(Empty))$

Flatten

Obtaining the string underlying a value:

$$
\begin{array}{lcl}
|Empty| & \stackrel{\text{def}}{=} & [] \\
|Char(c)| & \stackrel{\text{def}}{=} & [c] \\
|Left(v)| & \stackrel{\text{def}}{=} & |v| \\
|Right(v)| & \stackrel{\text{def}}{=} & |v| \\
|Seq(v_1, v_2)| & \stackrel{\text{def}}{=} & |v_1| \mathbin{@} |v_2| \\
|Stars\,[v_1, \ldots, v_n]| & \stackrel{\text{def}}{=} & |v_1| \mathbin{@} \ldots \mathbin{@} |v_n|
\end{array}
$$

r_1: $a \cdot (b \cdot c)$
r_2: $\mathbf{1} \cdot (b \cdot c)$
r_3: $(\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$
r_4: $(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$



v_1: $Seq(Char(a), Seq(Char(b), Char(c)))$
v_2: $Seq(Empty, Seq(Char(b), Char(c)))$
v_3: $Right(Seq(Empty, Char(c)))$
v_4: $Right(Right(Empty))$

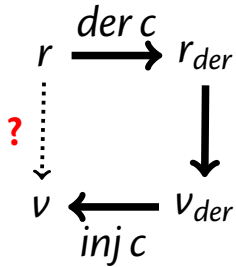
$|v_1|$: abc
$|v_2|$: bc
$|v_3|$: c
$|v_4|$: $[]$

Mkeps

Finding a (posix) value for recognising the empty string:

$$
\begin{array}{lcl}
mkeps\,(\mathbf{1}) & \stackrel{\text{def}}{=} & Empty \\
mkeps\,(r_1 + r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } Left(mkeps(r_1)) \\
 & & \text{else } Right(mkeps(r_2)) \\
mkeps\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & Seq(mkeps(r_1), mkeps(r_2)) \\
mkeps\,(r^*) & \stackrel{\text{def}}{=} & Stars\,[]
\end{array}
$$

Inject



Inject

Injecting ("Adding") a character to a value

$$
\begin{array}{ll}
inj\ (c)\ c\ (Empty) & \stackrel{\text{def}}{=} Char\ c \\
inj\ (r_1 + r_2)\ c\ (Left(v)) & \stackrel{\text{def}}{=} Left(inj\ r_1\ c\ v) \\
inj\ (r_1 + r_2)\ c\ (Right(v)) & \stackrel{\text{def}}{=} Right(inj\ r_2\ c\ v) \\
inj\ (r_1 \cdot r_2)\ c\ (Seq(v_1, v_2)) & \stackrel{\text{def}}{=} Seq(inj\ r_1\ c\ v_1, v_2) \\
inj\ (r_1 \cdot r_2)\ c\ (Left(Seq(v_1, v_2))) & \stackrel{\text{def}}{=} Seq(inj\ r_1\ c\ v_1, v_2) \\
inj\ (r_1 \cdot r_2)\ c\ (Right(v)) & \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\ r_2\ c\ v) \\
inj\ (r^*)\ c\ (Seq(v, Stars\ vs)) & \stackrel{\text{def}}{=} Stars\ (inj\ r\ c\ v\ ::\ vs)
\end{array}
$$

inj: 1st arg $\mapsto$ a rexp; 2nd arg $\mapsto$ a character; 3rd arg $\mapsto$ a value
result $\mapsto$ a value

$$inj\,(c)\,c\,(Empty) \stackrel{\text{def}}{=} Char\,c$$

$$inj\ (r_1 + r_2)\ c\ (Left(v)) \stackrel{\text{def}}{=} Left(inj\ r_1\ c\ v)$$
$$inj\ (r_1 + r_2)\ c\ (Right(v)) \stackrel{\text{def}}{=} Right(inj\ r_2\ c\ v)$$

$$
\begin{array}{ll}
inj\ (r_1 \cdot r_2)\ c\ (Seq(v_1, v_2)) & \stackrel{\text{def}}{=} Seq(inj\ r_1\ c\ v_1, v_2) \\
inj\ (r_1 \cdot r_2)\ c\ (Left(Seq(v_1, v_2))) & \stackrel{\text{def}}{=} Seq(inj\ r_1\ c\ v_1, v_2) \\
inj\ (r_1 \cdot r_2)\ c\ (Right(v)) & \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\ r_2\ c\ v)
\end{array}
$$

$$
der\ c\ (r_1 \cdot r_2) \stackrel{\text{def}}{=} \textbf{if}\ nullable(r_1)\ \textbf{then}\ (der\ c\ r_1) \cdot r_2 + der\ c\ r_2\ \textbf{else}\ (der\ c\ r_1) \cdot r_2
$$

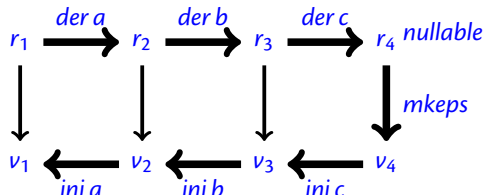
$$inj\,(r^*)\,c\,(Seq(v, Stars\,vs)) \stackrel{\text{def}}{=} Stars\,(inj\,r\,c\,v\ ::\ vs)$$

Lexing

$$lex\ r\ [] \stackrel{\text{def}}{=} \text{if } nullable(r) \text{ then } mkeps(r) \text{ else } error$$

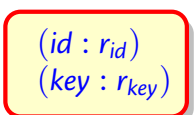
$$lex\ r\ a :: s \stackrel{\text{def}}{=} inj\ r\ a\ lex(der(a, r), s)$$

lex: returns a value



Records

- new regex: $(x : r)$ new value: $Rec(x, v)$



Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\, c\, (x : r) \stackrel{\text{def}}{=} der\, c\, r$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj\, (x : r)\, c\, v \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

$(id : r_{id})$
$(key : r_{key})$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\, c\, (x : r) \stackrel{\text{def}}{=} der\, c\, r$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj\, (x : r)\, c\, v \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

for extracting subpatterns $(z : ((x : ab) + (y : ba))$

$(id : r_{id})$
$(key : r_{key})$

- A regular expression for email addresses

$$\begin{array}{l}(\text{name: } [a\text{-}z0\text{-}9\_\,.-]^+)\cdot@\cdot\\ \quad(\text{domain: } [a\text{-}z0\text{-}9\,-]^+)\cdot.\cdot\\ \quad\quad(\text{top\_level: } [a\text{-}z\,.]^{\{2,6\}})\end{array}$$

`christian.urban@kcl.ac.uk`

- the result environment:

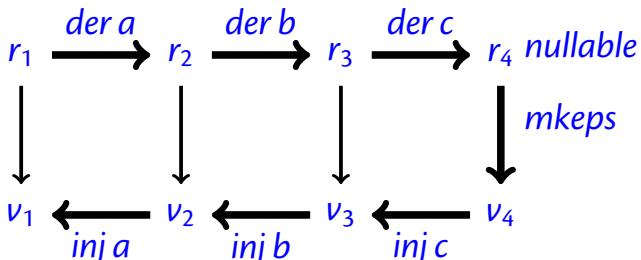
$$[(\mathit{name} : \texttt{christian.urban}),\ (\mathit{domain} : \texttt{kcl}),\ (\mathit{top\_level} : \texttt{ac.uk})]$$

While Tokens

$$\text{WHILE\_REGS} \stackrel{\text{def}}{=} (("k" : \text{KEYWORD}) + ("i" : \text{ID}) + ("o" : \text{OP}) + ("n" : \text{NUM}) + ("s" : \text{SEMI}) + ("p" : (\text{LPAREN} + \text{RPAREN})) + ("b" : (\text{BEGIN} + \text{END})) + ("w" : \text{WHITESPACE}))^*$$

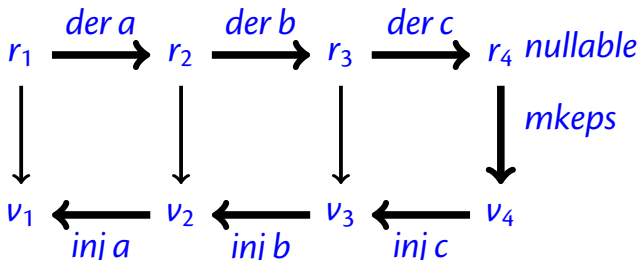
Simplification

- If we simplify after the derivative, then we are building the value for the simplified regular expression, but **not** for the original regular expression.



Simplification

- If we simplify after the derivative, then we are building the value for the simplified regular expression, but **not** for the original regular expression.



$$(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1}) \mapsto \mathbf{1}$$

Normally we would have

$$(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$$

and answer how this regular expression matches the empty string with the value

$$\mathit{Right}(\mathit{Right}(\mathit{Empty}))$$

But now we simplify this to $\mathbf{1}$ and would produce Empty (see *mkeps*).

Rectification

rectification functions:

$$
\begin{array}{lcll}
r \cdot \mathbf{0} & \mapsto & \mathbf{0} & \\
\mathbf{0} \cdot r & \mapsto & \mathbf{0} & \\
r \cdot \mathbf{1} & \mapsto & r & \lambda f_1\, f_2\, v.\, Seq(f_1\, v, f_2\, Empty) \\
\mathbf{1} \cdot r & \mapsto & r & \lambda f_1\, f_2\, v.\, Seq(f_1\, Empty, f_2\, v) \\
r + \mathbf{0} & \mapsto & r & \lambda f_1\, f_2\, v.\, Left(f_1\, v) \\
\mathbf{0} + r & \mapsto & r & \lambda f_1\, f_2\, v.\, Right(f_2\, v) \\
r + r & \mapsto & r & \lambda f_1\, f_2\, v.\, Left(f_1\, v)
\end{array}
$$

Rectification

			rectification functions:
$r \cdot \mathbf{0}$	$\mapsto$	$\mathbf{0}$	
$\mathbf{0} \cdot r$	$\mapsto$	$\mathbf{0}$	
$r \cdot \mathbf{1}$	$\mapsto$	r	$\lambda f_1\, f_2\, v.\, Seq(f_1\, v, f_2\, Empty)$
$\mathbf{1} \cdot r$	$\mapsto$	r	$\lambda f_1\, f_2\, v.\, Seq(f_1\, Empty, f_2\, v)$
$r + \mathbf{0}$	$\mapsto$	r	$\lambda f_1\, f_2\, v.\, Left(f_1\, v)$
$\mathbf{0} + r$	$\mapsto$	r	$\lambda f_1\, f_2\, v.\, Right(f_2\, v)$
$r + r$	$\mapsto$	r	$\lambda f_1\, f_2\, v.\, Left(f_1\, v)$

old *simp* returns a rexp;
new *simp* returns a rexp and a rectification function.

Rectification $\_ + \_$

$simp(r)$:

case $r = r_1 + r_2$

let $(r_{1s}, f_{1s}) = simp(r_1)$

$(r_{2s}, f_{2s}) = simp(r_2)$

case $r_{1s} = \mathbf{0}$: return $(r_{2s}, \lambda v.\, Right(f_{2s}(v)))$

case $r_{2s} = \mathbf{0}$: return $(r_{1s}, \lambda v.\, Left(f_{1s}(v)))$

case $r_{1s} = r_{2s}$: return $(r_{1s}, \lambda v.\, Left(f_{1s}(v)))$

otherwise: return $(r_{1s} + r_{2s}, f_{alt}(f_{1s}, f_{2s}))$

$f_{alt}(f_1, f_2) \stackrel{\text{def}}{=}$

$\lambda v.$ case $v = Left(v')$: return $Left(f_1(v'))$

case $v = Right(v')$: return $Right(f_2(v'))$

```
def simp(r: Rexp): (Rexp, Val => Val) = r match {
  case ALT(r1, r2) => {
    val (r1s, f1s) = simp(r1)
    val (r2s, f2s) = simp(r2)
    (r1s, r2s) match {
      case (ZERO, _) => (r2s, F_RIGHT(f2s))
      case (_, ZERO) => (r1s, F_LEFT(f1s))
      case _ =>
         if (r1s == r2s) (r1s, F_LEFT(f1s))
         else (ALT (r1s, r2s), F_ALT(f1s, f2s))
    }
  }
  ...
}
def F_RIGHT(f: Val => Val) = (v:Val) => Right(f(v))
def F_LEFT(f: Val => Val) = (v:Val) => Left(f(v))
def F_ALT(f1: Val => Val, f2: Val => Val) =
  (v:Val) => v match {
    case Right(v) => Right(f2(v))
    case Left(v) => Left(f1(v)) }
```

Rectification $\_ \cdot \_$

$simp(r)$:...

case $r = r_1 \cdot r_2$

let $(r_{1s}, f_{1s}) = simp(r_1)$

$(r_{2s}, f_{2s}) = simp(r_2)$

case $r_{1s} = \mathbf{0}$: return $(\mathbf{0}, f_{error})$

case $r_{2s} = \mathbf{0}$: return $(\mathbf{0}, f_{error})$

case $r_{1s} = \mathbf{1}$: return $(r_{2s}, \lambda v.\, Seq(f_{1s}(Empty), f_{2s}(v)))$

case $r_{2s} = \mathbf{1}$: return $(r_{1s}, \lambda v.\, Seq(f_{1s}(v), f_{2s}(Empty)))$

otherwise: return $(r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$

$f_{seq}(f_1, f_2) \stackrel{\text{def}}{=}$

$\lambda v.$ case $v = Seq(v_1, v_2)$: return $Seq(f_1(v_1), f_2(v_2))$

```
def simp(r: Rexp): (Rexp, Val => Val) = r match {
  case SEQ(r1, r2) => {
    val (r1s, f1s) = simp(r1)
    val (r2s, f2s) = simp(r2)
    (r1s, r2s) match {
      case (ZERO, _) => (ZERO, F_ERROR)
      case (_, ZERO) => (ZERO, F_ERROR)
      case (ONE, _) => (r2s, F_SEQ_Empty1(f1s, f2s))
      case (_, ONE) => (r1s, F_SEQ_Empty2(f1s, f2s))
      case _ => (SEQ(r1s,r2s), F_SEQ(f1s, f2s))
    }
  }
  ...}
def F_SEQ_Empty1(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(Empty), f2(v))
def F_SEQ_Empty2(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(v), f2(Empty))
def F_SEQ(f1: Val => Val, f2: Val => Val) =
  (v:Val) => v match {
    case Sequ(v1, v2) => Sequ(f1(v1), f2(v2)) }
```

Rectification Example

$$(b \cdot c) + (\mathbf{0} + \mathbf{1}) \mapsto (b \cdot c) + \mathbf{1}$$

Rectification Example

$$\underline{(b \cdot c)} + (\underline{\mathbf{0} + \mathbf{1}}) \mapsto (b \cdot c) + \mathbf{1}$$

Rectification Example

$$(\underline{b \cdot c}) + (\underline{\mathbf{0} + \mathbf{1}}) \mapsto (b \cdot c) + \mathbf{1}$$

$$\begin{aligned} f_{s1} &= \lambda v.v \\ f_{s2} &= \lambda v.Right(v) \end{aligned}$$

Rectification Example

$$\underline{(b \cdot c) + (\mathbf{0} + \mathbf{1})} \mapsto (b \cdot c) + \mathbf{1}$$

$$\begin{array}{lcl} f_{s1} & = & \lambda v.v \\ f_{s2} & = & \lambda v.Right(v) \end{array}$$

$$f_{alt}(f_{s1}, f_{s2}) \stackrel{\text{def}}{=}$$
$$\lambda v.\ \text{case}\ v = Left(v'):\ \ \text{return}\ Left(f_{s1}(v'))$$
$$\text{case}\ v = Right(v'):\ \text{return}\ Right(f_{s2}(v'))$$

Rectification Example

$$\underline{(b \cdot c) + (\mathbf{0} + \mathbf{1})} \mapsto (b \cdot c) + \mathbf{1}$$

$$\begin{array}{lcl} f_{s1} & = & \lambda v.v \\ f_{s2} & = & \lambda v.Right(v) \end{array}$$

$$\begin{array}{ll} \lambda v. & \text{case } v = Left(v'): \quad \text{return } Left(v') \\ & \text{case } v = Right(v'): \text{return } Right(Right(v')) \end{array}$$

Rectification Example

$$\underline{(b \cdot c) + (\mathbf{0} + \mathbf{1})} \mapsto (b \cdot c) + \mathbf{1}$$

$$\begin{array}{lcl} f_{s1} & = & \lambda v.v \\ f_{s2} & = & \lambda v.Right(v) \end{array}$$

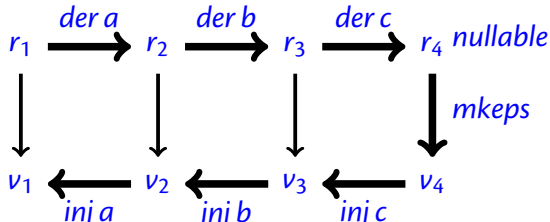
$$\begin{array}{l} \lambda v.\ \text{case } v = Left(v'): \quad \text{return } Left(v') \\ \qquad\ \text{case } v = Right(v'): \text{return } Right(Right(v')) \end{array}$$

mkeps simplified case: $Right(Empty)$
rectified case: $Right(Right(Empty))$

Lexing with Simplification

$$lex\ r\ [] \stackrel{def}{=} \text{if } nullable(r) \text{ then } mkeps(r) \text{ else } error$$

$$lex\ r\ c :: s \stackrel{def}{=} \text{let } (r', frect) = simp(der(c, r))$$
$$inj\ r\ c\ (frect(lex(r', s)))$$



Environments

Obtaining the "recorded" parts of a value:

$$
\begin{array}{lcl}
env(Empty) & \stackrel{\text{def}}{=} & [] \\
env(Char(c)) & \stackrel{\text{def}}{=} & [] \\
env(Left(v)) & \stackrel{\text{def}}{=} & env(v) \\
env(Right(v)) & \stackrel{\text{def}}{=} & env(v) \\
env(Seq(v_1, v_2)) & \stackrel{\text{def}}{=} & env(v_1) \mathbin{@} env(v_2) \\
env(Stars\,[v_1, \ldots, v_n]) & \stackrel{\text{def}}{=} & env(v_1) \mathbin{@} \ldots \mathbin{@} env(v_n) \\
env(Rec(x : v)) & \stackrel{\text{def}}{=} & (x : |v|) :: env(v)
\end{array}
$$

While Tokens

$$\texttt{WHILE\_REGS} \stackrel{\text{def}}{=} (\texttt{("k" : KEYWORD)} + \texttt{("i" : ID)} + \texttt{("o" : OP)} + \texttt{("n" : NUM)} + \texttt{("s" : SEMI)} + \texttt{("p" : (LPAREN + RPAREN))} + \texttt{("b" : (BEGIN + END))} + \texttt{("w" : WHITESPACE)})^*$$

```
"if true then then 42 else +"
```

```
KEYWORD(if),
WHITESPACE,
IDENT(true),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
NUM(42),
WHITESPACE,
KEYWORD(else),
WHITESPACE,
OP(+)
```

```
"if true then then 42 else +"
```

```
KEYWORD(if),
IDENT(true),
KEYWORD(then),
KEYWORD(then),
NUM(42),
KEYWORD(else),
OP(+)
```

Lexer: Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as next token.
- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

Environments

Obtaining the "recorded" parts of a value:

$$
\begin{array}{lcl}
env(Empty) & \stackrel{\text{def}}{=} & [] \\
env(Char(c)) & \stackrel{\text{def}}{=} & [] \\
env(Left(v)) & \stackrel{\text{def}}{=} & env(v) \\
env(Right(v)) & \stackrel{\text{def}}{=} & env(v) \\
env(Seq(v_1, v_2)) & \stackrel{\text{def}}{=} & env(v_1) \mathbin{@} env(v_2) \\
env(Stars[v_1, \ldots, v_n]) & \stackrel{\text{def}}{=} & env(v_1) \mathbin{@} \ldots \mathbin{@} env(v_n) \\
env(Rec(x : v)) & \stackrel{\text{def}}{=} & (x : |v|) :: env(v)
\end{array}
$$

While Tokens

$$\texttt{WHILE\_REGS} \stackrel{\text{def}}{=} (\texttt{("k" : KEYWORD)} + \texttt{("i" : ID)} + \texttt{("o" : OP)} + \texttt{("n" : NUM)} + \texttt{("s" : SEMI)} + \texttt{("p" : (LPAREN + RPAREN))} + \texttt{("b" : (BEGIN + END))} + \texttt{("w" : WHITESPACE)})^*$$

```
"if true then then 42 else +"
```

```
KEYWORD(if),
WHITESPACE,
IDENT(true),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
NUM(42),
WHITESPACE,
KEYWORD(else),
WHITESPACE,
OP(+)
```

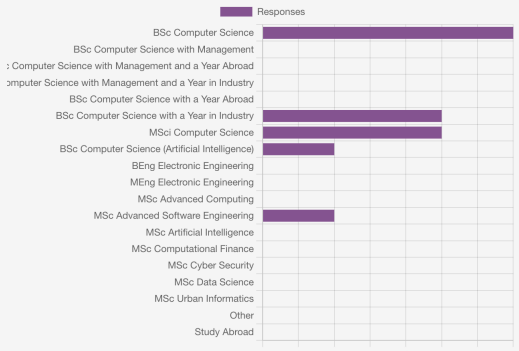
```
"if true then then 42 else +"
```

```
KEYWORD(if),
IDENT(true),
KEYWORD(then),
KEYWORD(then),
NUM(42),
KEYWORD(else),
OP(+)
```

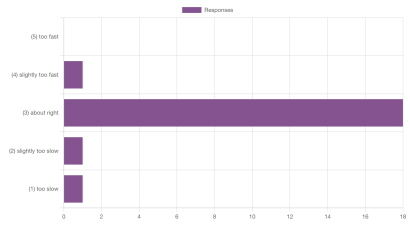
Week 3 Feedback

Submitted answers: 21
Questions: 12

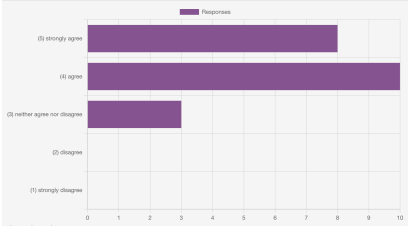
(Programme) Which degree programme are you studying?



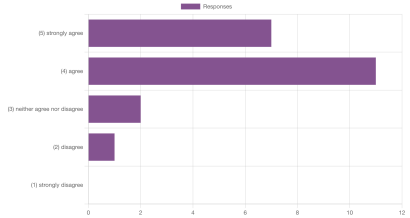
(AppropriatePace) ...teaches at a pace that is:



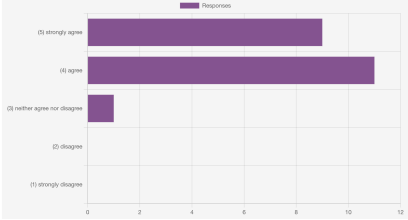
(ExplainsMaterialClearly) ...explains the material clearly



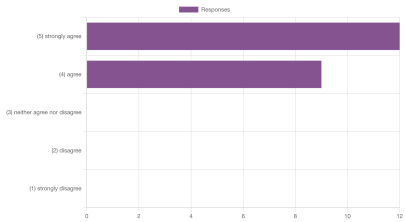
(contemporary) ...makes clear the contemporary relevance of the subject



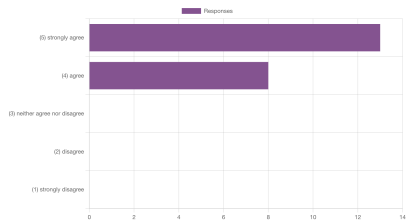
(keats) ...provides useful information on KEATS



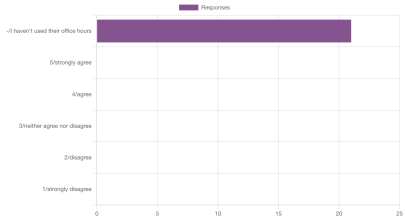
(objectives) ...has (have) made the module objectives clear



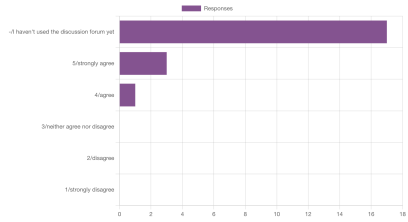
(amethods) ...has (have) made the assessment methods clear



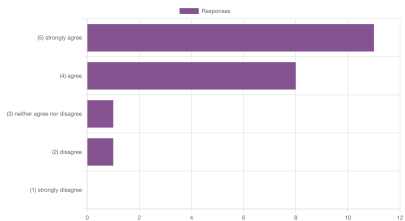
(ohours) ...is available to answer questions in office hours:



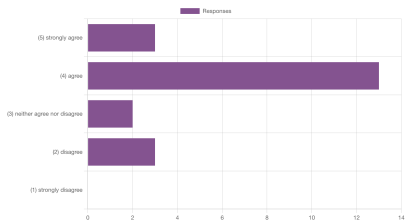
(forum) ...is available to answer questions on the discussion forum:



(Audible) The video lectures and other content on KEATS are helpful



(facilities) The live teaching sessions are helpful

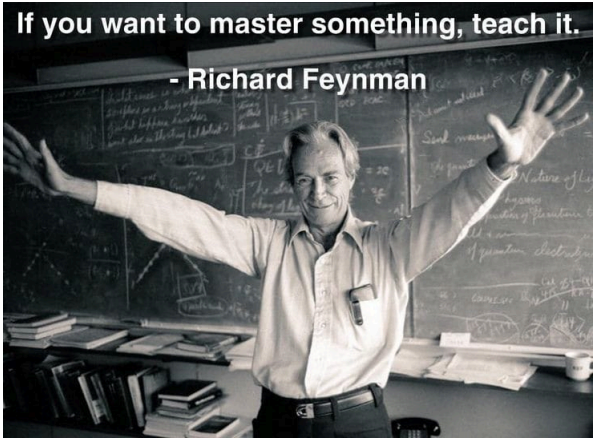


- Thank you for your hand out. I took your advice and finally read the ho4 before the lecture videos. It was very smooth to read, but at times it was hard to understand. But knowing my friendly lecturer would explain the concepts in the videos I carried on. Although it was a bit painful, new information always is, and a bit boring at times, it's probably my bad attention span. I finished it and found it quite fun, probably the most approachable and fun technical handout I've read so far. Reading it before the videos encouraged me to stay on it because I didn't assume I understood the content and the anticipation of the unknown was fun.

- things are going well initially, but in week3 i am really lost, each 30 mins video took me hours to understand, things are too abstract, wish there are more explanations
- The LGTs kind of just repeat the information from the lectures. I think they would be more beneficial if you could explain certain topics in more detail and go through more examples.
- The room is somehow always either too warm or too cold.
- Module content is enjoyable and interesting. Even with difficult part of contents, Dr Urban provides clear explanations.

- The in person lectures could be a bit faster, I think more focus on questions would be useful rather than repeating the videos.
- The teacher explains the material very well and is able to answer most questions during the large group tutorials. I really like the teacher's enthusiasm; throwing in little jokes from time to time and referring to the history of compilers makes me more interested in the subject and inspires me to delve deeper into it. Since I am an MSc student, I did not take any previous Level 6 modules, which the teacher refers to quite often. However, the teacher provides enough basic knowledge, even if you haven't learned about these topics before. The module is conducted using a flipped-classroom system, so the 2-hour lecture every Friday is spent 'revising' the material that was provided in the videos. Personally, I like this approach as it helps reinforce important aspects of both current and previous lessons.

- Would appreciate some time in the LGTs to go over specific examples of questions from the slides.
- The recorded content is too long. Also, the Sgt room (Monday 2-3 with Harry) is really small compared to the students who show up.
- hopefully there will be solution or any tutorials / lectures for cw1
- Normally, I wouldn't have such an audacious request as this. However, the lecture hall we were assigned is absoluely terrible; this is mainly because indivduals at the back are so far away from the front that they think they can get away with talking loudly and being distracting. If that isn't possible, then can the wall sockets on the right side at least work.


If you want to master something, teach it.
- Richard Feynman