Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk
Slides & Progs: KEATS (also homework is there)

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

The Goal of this Course
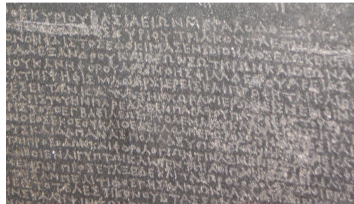
Write a compiler

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 · (LETTER + DIGIT + _)*
NUM:
  (NONZERODIGIT · DIGIT*) + 0
OP:
  +, -, *, %, <, <=
COMMENT:
  /* · ~(ALL* · (*/) · ALL*) · */
```

```
  "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
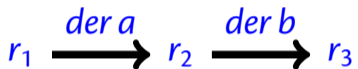
traditional lexers are fast, but hairy

Sulzmann & Lu Matcher
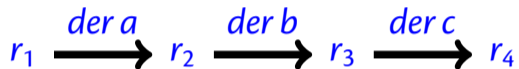
We want to match the string *abc* using r_1:

$$r_1 \xrightarrow{\text{der } a} r_2$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{\ der\ a\ } r_2 \xrightarrow{\ der\ b\ } r_3$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{\text{der } a} r_2 \xrightarrow{\text{der } b} r_3 \xrightarrow{\text{der } c} r_4$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{\text{der } a} r_2 \xrightarrow{\text{der } b} r_3 \xrightarrow{\text{der } c} r_4 \text{ nullable?}$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{der\ a} r_2 \xrightarrow{der\ b} r_3 \xrightarrow{der\ c} r_4\ nullable?$$

$$r_4 \downarrow$$

$$v_4$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{\textit{der a}} r_2 \xrightarrow{\textit{der b}} r_3 \xrightarrow{\textit{der c}} r_4 \ \textit{nullable?}$$

$$v_3 \xleftarrow{\textit{inj c}} v_4 \longleftarrow r_4$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{\textit{der a}} r_2 \xrightarrow{\textit{der b}} r_3 \xrightarrow{\textit{der c}} r_4 \quad \textit{nullable?}$$

$$v_2 \xleftarrow{\textit{inj b}} v_3 \xleftarrow{\textit{inj c}} v_4 \quad \downarrow$$

Sulzmann & Lu Matcher

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

$$r_1 \xrightarrow{\text{der } a} r_2 \xrightarrow{\text{der } b} r_3 \xrightarrow{\text{der } c} r_4 \; \text{nullable?}$$

$$v_1 \xleftarrow{\text{inj } a} v_2 \xleftarrow{\text{inj } b} v_3 \xleftarrow{\text{inj } c} v_4$$
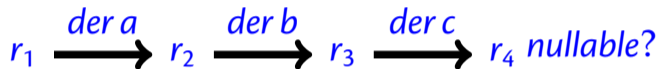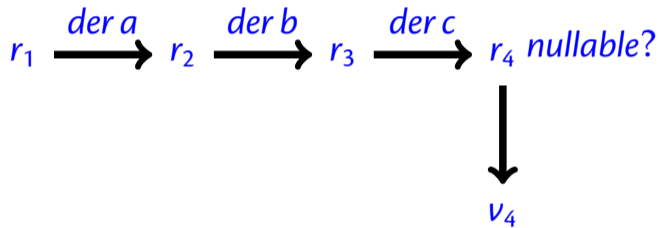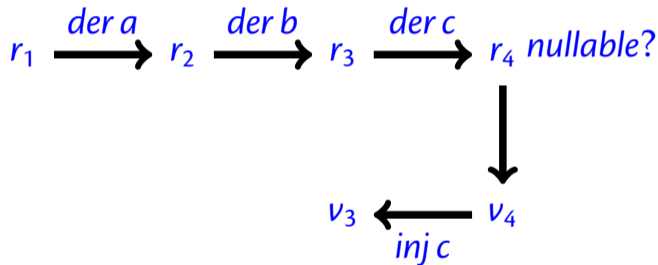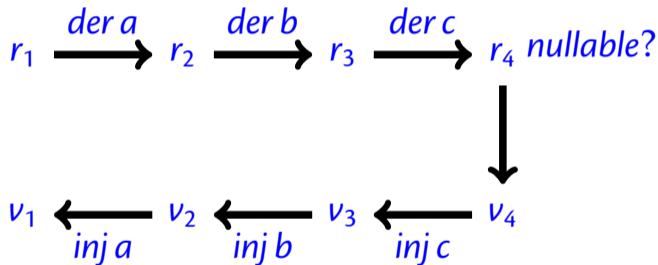
Sulzmann & Lu Matcher

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

Regexes and Values

Regular expressions and their corresponding values:

$$r ::= \mathbf{0}$$
$$| \quad \mathbf{1}$$
$$| \quad c$$
$$| \quad r_1 \cdot r_2$$
$$| \quad r_1 + r_2$$
$$| \quad r^*$$

$$v ::= $$
$$Empty$$
$$| \quad Char(c)$$
$$| \quad Seq(v_1, v_2)$$
$$| \quad Left(v)$$
$$| \quad Right(v)$$
$$| \quad Stars\,[\,]$$
$$| \quad Stars\,[v_1, \ldots v_n]$$

```scala
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
```
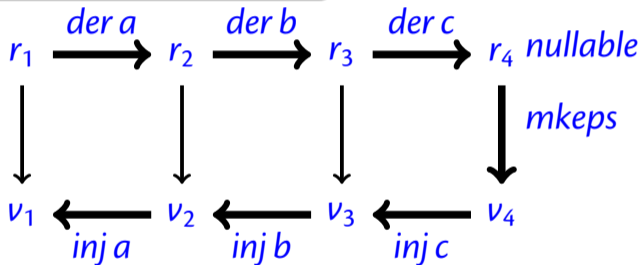
r_1: $a \cdot (b \cdot c)$
r_2: $\mathbf{1} \cdot (b \cdot c)$
r_3: $(\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$
r_4: $(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$
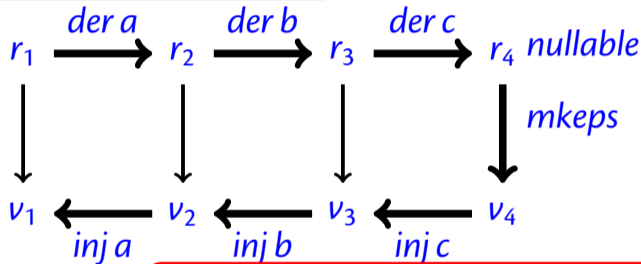
$r_1 \xrightarrow{\text{der } a} r_2 \xrightarrow{\text{der } b} r_3 \xrightarrow{\text{der } c} r_4$ *nullable*

$r_4 \xrightarrow{\text{mkeps}} v_4$

$v_1 \xleftarrow{\text{inj } a} v_2 \xleftarrow{\text{inj } b} v_3 \xleftarrow{\text{inj } c} v_4$

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

$r_1 \xrightarrow{\ der\ a\ } r_2 \xrightarrow{\ der\ b\ } r_3 \xrightarrow{\ der\ c\ } r_4$ nullable

$r_4 \xrightarrow{\ mkeps\ } v_4$

$v_1 \xleftarrow{\ inj\ a\ } v_2 \xleftarrow{\ inj\ b\ } v_3 \xleftarrow{\ inj\ c\ } v_4$

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:

$$|Empty| \overset{\text{def}}{=} []$$
$$|Char(c)| \overset{\text{def}}{=} [c]$$
$$|Left(v)| \overset{\text{def}}{=} |v|$$
$$|Right(v)| \overset{\text{def}}{=} |v|$$
$$|Seq(v_1, v_2)| \overset{\text{def}}{=} |v_1| @ |v_2|$$
$$|Stars\,[v_1, \ldots, v_n]| \overset{\text{def}}{=} |v_1| @ \ldots @ |v_n|$$

$$r_1: \quad a \cdot (b \cdot c)$$
$$r_2: \quad \mathbf{1} \cdot (b \cdot c)$$
$$r_3: \quad (\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$$
$$r_4: \quad (\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$$
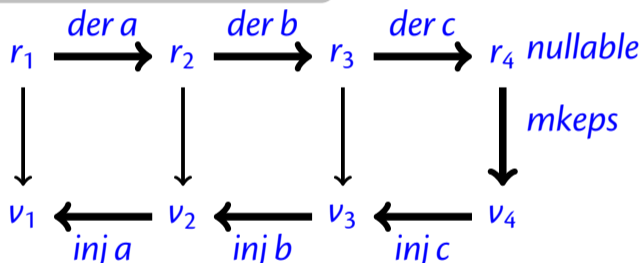
$r_1 \xrightarrow{der\ a} r_2 \xrightarrow{der\ b} r_3 \xrightarrow{der\ c} r_4$ *nullable*

r_4 *mkeps*

$v_1 \xleftarrow{inj\ a} v_2 \xleftarrow{inj\ b} v_3 \xleftarrow{inj\ c} v_4$

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

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

Mkeps

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

$$mkeps\ (\mathbf{1}) \stackrel{def}{=} Empty$$

$$mkeps\ (r_1 + r_2) \stackrel{def}{=} \text{if } nullable(r_1)$$
$$\qquad\qquad\qquad \text{then } Left(mkeps(r_1))$$
$$\qquad\qquad\qquad \text{else } Right(mkeps(r_2))$$

$$mkeps\ (r_1 \cdot r_2) \stackrel{def}{=} Seq(mkeps(r_1), mkeps(r_2))$$

$$mkeps\ (r^*) \stackrel{def}{=} Stars\ []$$

Inject
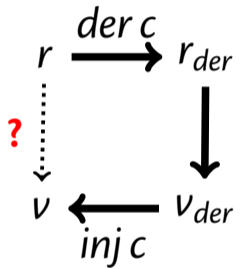
Inject

Injecting ("Adding") a character to a value

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

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

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

$$inj\,(r_1 \cdot r_2)\,c\,(Seq(v_1, v_2)) \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2)$$

$$inj\,(r_1 \cdot r_2)\,c\,(Left(Seq(v_1, v_2))) \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2)$$

$$inj\,(r_1 \cdot r_2)\,c\,(Right(v)) \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\,r_2\,c\,v)$$

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

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

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

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

$inj\ (r_1 \cdot r_2)\ c\ (Seq(v_1, v_2)) \stackrel{\text{def}}{=} Seq(inj\ r_1\ c\ v_1, v_2)$

$inj\ (r_1 \cdot r_2)\ c\ (Left(Seq(v_1, v_2))) \stackrel{\text{def}}{=} Seq(inj\ r_1\ c\ v_1, v_2)$

$inj\ (r_1 \cdot r_2)\ c\ (Right(v)) \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\ r_2\ c\ v)$

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

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

Lexing
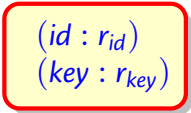
$$lex\ r\ [] \quad \overset{\text{def}}{=} \text{if } nullable(r) \text{ then } mkeps(r) \text{ else } error$$
$$lex\ r\ a :: s \overset{\text{def}}{=} inj\ r\ a\ lex(der(a, r), s)$$

lex: returns a value
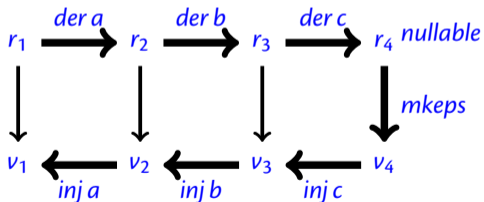
Records

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

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

Records

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

$nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$

$der\, c\, (x : r) \stackrel{\text{def}}{=} der\, c\, r$

$mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$

$inj\, (x : r)\, c\, v \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

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

Records

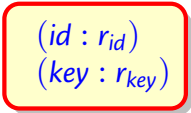new regex: $(x : r)$ new value: $Rec(x, v)$

$$nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$$

$$der\, c\, (x : r) \stackrel{\text{def}}{=} der\, c\, r$$

$$mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$$

$$inj\, (x : r)\, c\, v \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$$

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

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

A regular expression for email addresses

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

```
christian.urban@kcl.ac.uk
```

the result environment:

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

While Tokens

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

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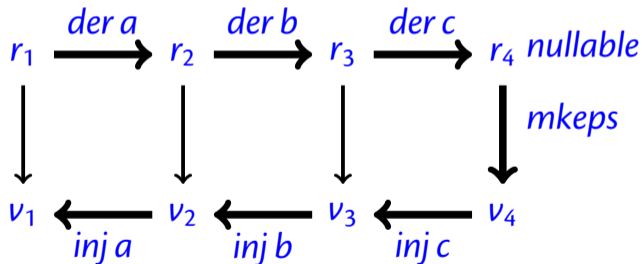
$$(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1}) \mapsto \mathbf{1}$$

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

$$Right(Right(Empty))$$

But now we simplify this to **1** and would produce *Empty* (see *mkeps*).

Rectification

rectification
functions:

$$r \cdot \mathbf{0} \;\mapsto\; \mathbf{0}$$

$$\mathbf{0} \cdot r \;\mapsto\; \mathbf{0}$$

$$r \cdot \mathbf{1} \;\mapsto\; r \qquad \lambda f_1 f_2 \, v. \, Seq(f_1 \, v, f_2 \, Empty)$$

$$\mathbf{1} \cdot r \;\mapsto\; r \qquad \lambda f_1 f_2 \, v. \, Seq(f_1 \, Empty, f_2 \, v)$$

$$r + \mathbf{0} \;\mapsto\; r \qquad \lambda f_1 f_2 \, v. \, Left(f_1 \, v)$$

$$\mathbf{0} + r \;\mapsto\; r \qquad \lambda f_1 f_2 \, v. \, Right(f_2 \, v)$$

$$r + r \;\mapsto\; r \qquad \lambda f_1 f_2 \, v. \, Left(f_1 \, v)$$

Rectification

rectification
functions:

$$r \cdot \mathbf{0} \;\mapsto\; \mathbf{0}$$

$$\mathbf{0} \cdot r \;\mapsto\; \mathbf{0}$$

$$r \cdot \mathbf{1} \;\mapsto\; r \qquad \lambda f_1 f_2\, v.\, Seq(f_1\, v, f_2\, Empty)$$

$$\mathbf{1} \cdot r \;\mapsto\; r \qquad \lambda f_1 f_2\, v.\, Seq(f_1\, Empty, f_2\, v)$$

$$r + \mathbf{0} \;\mapsto\; r \qquad \lambda f_1 f_2\, v.\, Left(f_1\, v)$$

$$\mathbf{0} + r \;\mapsto\; r \qquad \lambda f_1 f_2\, v.\, Right(f_2\, v)$$

$$r + r \;\mapsto\; r \qquad \lambda f_1 f_2\, v.\, Left(f_1\, v)$$

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 $Right(f_2(v'))$

```scala
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 _ · _

$simp(r)$:...
   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 $(r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$

  $f_{seq}(f_1, f_2) \overset{\text{def}}{=}$
    $\lambda v.\ $case $v = Seq(v_1, v_2)$: return $Seq(f_1(v_1), f_2(v_2))$

```scala
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

$$(b \cdot c) + (0 + 1) \mapsto (b \cdot c) + 1$$

Rectification Example

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

Rectification Example

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

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

Rectification Example

$$(b \cdot c) + (0 + 1) \mapsto (b \cdot c) + 1$$

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

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

Rectification Example

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

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

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

Rectification Example

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

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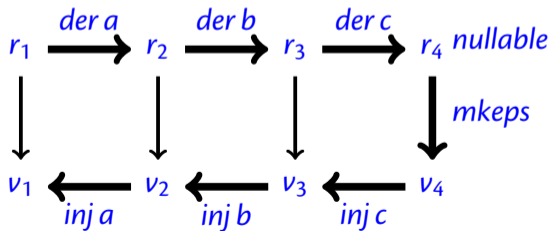f_{s2} &= \lambda v.Right(v)
\end{aligned}
$$

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

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

Lexing with Simplification

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

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

$$inj\ r\ c\ (frect(lex(r', s)))$$

Environments

Obtaining the "recorded" parts of a value:

$$env(Empty) \stackrel{\text{def}}{=} []$$

$$env(Char(c)) \stackrel{\text{def}}{=} []$$

$$env(Left(v)) \stackrel{\text{def}}{=} env(v)$$

$$env(Right(v)) \stackrel{\text{def}}{=} env(v)$$

$$env(Seq(v_1, v_2)) \stackrel{\text{def}}{=} env(v_1) \mathbin{@} env(v_2)$$

$$env(Stars\,[v_1, \ldots, v_n]) \stackrel{\text{def}}{=} env(v_1) \mathbin{@} \ldots \mathbin{@} env(v_n)$$

$$env(Rec(x : v)) \stackrel{\text{def}}{=} (x : |v|) :: env(v)$$

While Tokens

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

```
      "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:

$$env(Empty) \quad\stackrel{\text{def}}{=}\quad []$$

$$env(Char(c)) \quad\stackrel{\text{def}}{=}\quad []$$

$$env(Left(v)) \quad\stackrel{\text{def}}{=}\quad env(v)$$

$$env(Right(v)) \quad\stackrel{\text{def}}{=}\quad env(v)$$

$$env(Seq(v_1, v_2)) \quad\stackrel{\text{def}}{=}\quad env(v_1) @ env(v_2)$$

$$env(Stars[v_1, \ldots, v_n]) \quad\stackrel{\text{def}}{=}\quad env(v_1) @ \ldots @ env(v_n)$$

$$env(Rec(x : v)) \quad\stackrel{\text{def}}{=}\quad (x : |v|) :: env(v)$$

While Tokens

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

```
        "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(+)
```

Are dfas completed by definition as in do they have a to have transitions for every char at every state?

How can you tell if a language will be regular or irregular?

- contentwise probably the most enjoyable module I have had so far at KCL

- I personally found the coursework sheets a bit unclear. For example I couldnt see what was required from the CFUN section but once explained it actually was very easy and didnt take long to get working

- Please can tutorial sessions be recorded & linked on Keats.

- One of the best taught modules I've had, with a genuinely interested and engaging lecturer. Thanks Dr. Urban!

- Dr. Urban is honestly a great lecturer, he's incredibly helpful and responsive. He also teaches at a very good pace and explains things clearly so students who sometimes struggle with the content like myself can keep up. It's a pleasure to learn from him.

- I just wish the Coursework content was explained better, in such a way that allows students to get on with the coursework as soon as possible.

- I'm thoroughly enjoying the module so far. I find that the handouts and homework really solidify my knowledge and the feedback is extremely useful. I enjoy doing the homework and then using the feedback alongside the workshop to correct where I have gone wrong.

- I enjoy this module and think it is taught well. The homework is very useful.
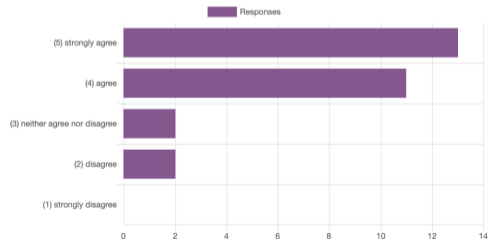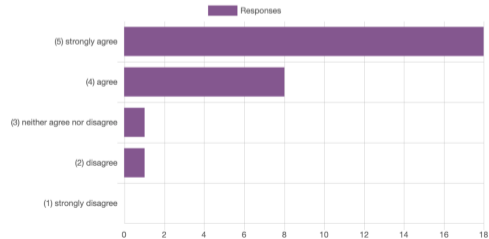
(AppropriatePace) ...teaches at a pace that is:
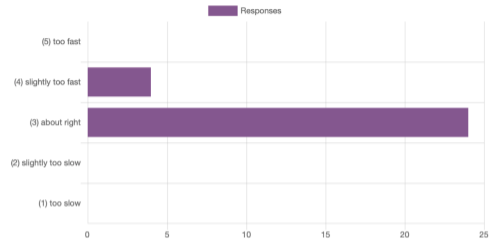
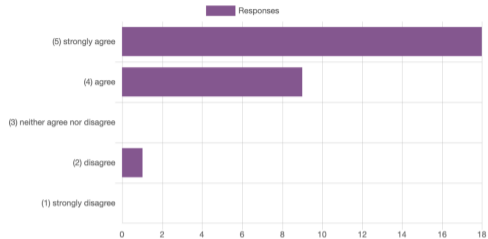**(ExplainsMaterialClearly) ...explains the material clearly**
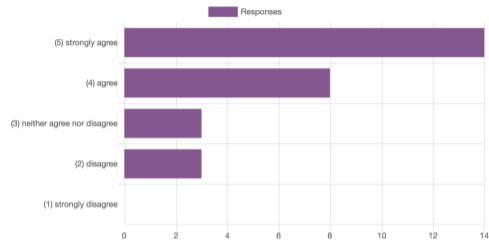
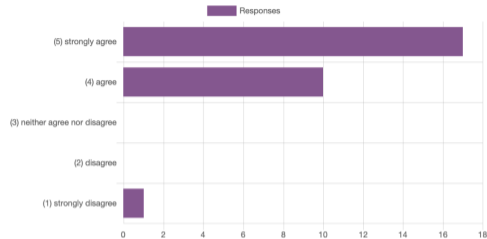**(contemporary) ..makes clear the contemporary relevance of the subject**
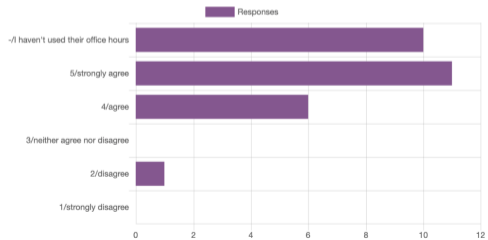

(keats) ...provides useful information on KEATS

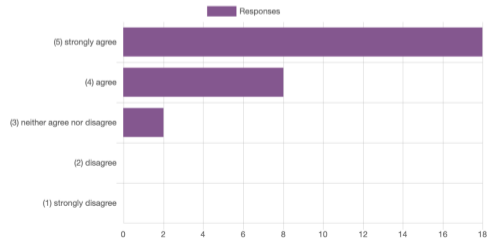## (objectives) ...has (have) made the module objectives clear

Responses

- (5) strongly agree
- (4) agree
- (3) neither agree nor disagree
- (2) disagree
- (1) strongly disagree

0 2 4 6 8 10 12 14 16 18

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

Responses

- (5) strongly agree
- (4) agree
- (3) neither agree nor disagree
- (2) disagree
- (1) strongly disagree

0 2 4 6 8 10 12 14 16 18

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

Responses

- -/I haven't used their office hours
- 5/strongly agree
- 4/agree
- 3/neither agree nor disagree
- 2/disagree
- 1/strongly disagree

0 2 4 6 8 10 12

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

Responses

- (5) strongly agree
- (4) agree
- (3) neither agree nor disagree
- (2) disagree
- (1) strongly disagree

0 2 4 6 8 10 12 14 16 18

(facilities) The live teaching sessions are helpful

Legend: Responses

Response	Count
(5) strongly agree	15
(4) agree	7
(3) neither agree nor disagree	4
(2) disagree	1
(1) strongly disagree	1