

Compilers and Formal Languages (4)

Email: christian.urban at kcl.ac.uk

Office: N7.07 (North Wing, Bush House)

Slides: KEATS (also homework is there)

Survey: Thanks!

"...Thanks a million! Thanks without end!"



*"Urban is a very talented lecturer:
thorough, concise, clear, and cares to make
sure that we are learning!"*

Survey: Thanks!

"...Thanks a million! Thanks without end!"



*"Urban is a very talented lecturer:
thorough, concise, clear, and cares to make
sure that we are learning!"*

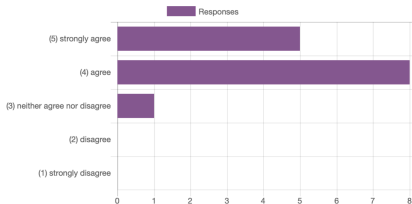
Survey: Thanks!

“...Thanks a million! Thanks without end!”

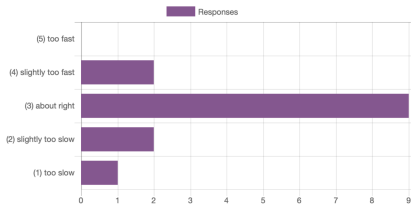


*“Urban is a very talented lecturer:
thorough, concise, clear, and cares to make
sure that we are learning!”*

(Audible) ...is (are) audible



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



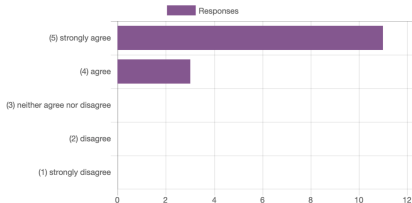
Survey: Thanks!

“...Thanks a million! Thanks without end!”

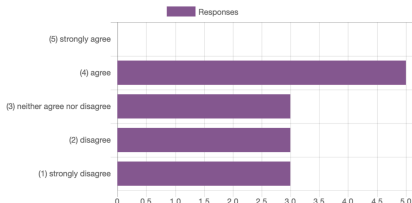


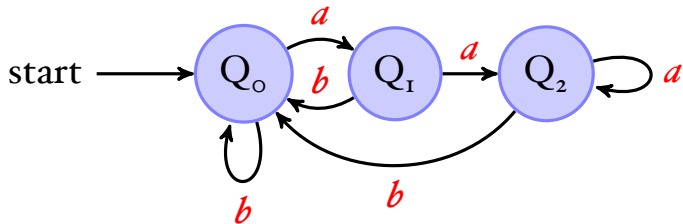
*“Urban is a very talented lecturer:
thorough, concise, clear, and cares to make
sure that we are learning!”*

(ExplainsMaterialClearly) ...explains the material clearly



(facilities) The facilities and room function well





$$\begin{aligned}
 Q_0 &= \mathbf{I} + Q_0 b + Q_I b + Q_2 b \\
 Q_I &= Q_0 a \\
 Q_2 &= Q_I a + Q_2 a
 \end{aligned}$$

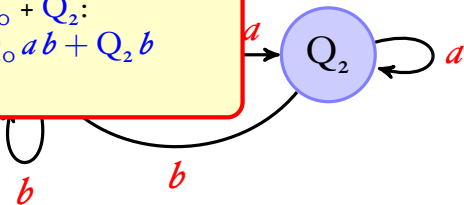
Arden's Lemma:

$$\text{If } q = qr + s \text{ then } q = sr^*$$

substitute Q_I into $Q_0 + Q_2$:

$$Q_0 = \mathbf{I} + Q_0 b + Q_0 a b + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$



$$Q_0 = \mathbf{I} + Q_0 b + Q_I b + Q_2 b$$

$$Q_I = Q_0 a$$

$$Q_2 = Q_I a + Q_2 a$$

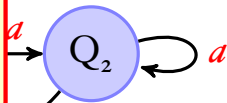
Arden's Lemma:

$$\text{If } q = qr + s \text{ then } q = sr^*$$

substitute Q_I into $Q_0 + Q_2$:

$$Q_0 = \mathbf{I} + Q_0 b + Q_0 a b + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$



simplifying Q_0 :

$$Q_0 = \mathbf{I} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$

$$Q_0 = \mathbf{I} + Q_0 b + Q_I b + Q_2 b$$

$$Q_I = Q_0 a$$

$$Q_2 = Q_I a + Q_2 a$$

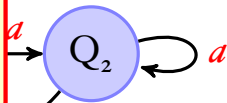
Arden's Lemma:

$$\text{If } q = qr + s \text{ then } q = sr^*$$

substitute Q_1 into $Q_0 + Q_2$:

$$Q_0 = \mathbf{1} + Q_0 b + Q_0 a b + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$



simplifying Q_0 :

$$Q_0 = \mathbf{1} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$

Arden for Q_2 :

$$Q_0 = \mathbf{1} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a (a^*)$$

$$Q_2 = Q_1 a + Q_2 a$$

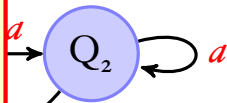
Arden's Lemma:

If $q = q r + s$ then $q = s r^*$

substitute Q_1 into $Q_0 + Q_2$:

$$Q_0 = \mathbf{1} + Q_0 b + Q_0 a b + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$



simplifying Q_0 :

$$Q_0 = \mathbf{1} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$

Arden for Q_2 :

$$Q_0 = \mathbf{1} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a (a^*)$$

Substitute Q_2 and simplify:

$$Q_0 = \mathbf{1} + Q_0 (b + a b + a a (a^*) b)$$

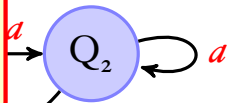
Arden's Lemma

If $q = q r + s$ then $q = s r^*$

substitute Q_1 into $Q_0 + Q_2$:

$$Q_0 = \mathbf{1} + Q_0 b + Q_0 a b + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$



simplifying Q_0 :

$$Q_0 = \mathbf{1} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a + Q_2 a$$

Arden for Q_2 :

$$Q_0 = \mathbf{1} + Q_0 (b + a b) + Q_2 b$$

$$Q_2 = Q_0 a a (a^*)$$

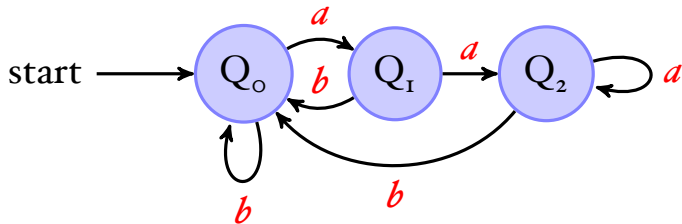
Substitute Q_2 and simplify:

$$Q_0 = \mathbf{1} + Q_0 (b + a b + a a (a^*) b)$$

Arden's Lemma

If Arden again for Q_0 :

$$Q_0 = (b + a b + a a (a^*) b)^*$$



$$Q_0 = \mathbf{I} + Q_0 b + Q_I b + Q_2 b$$

$$Q_I = Q_0 a$$

$$Q_2 = Q_I a + Q_2 a$$

Arden's Lemma:

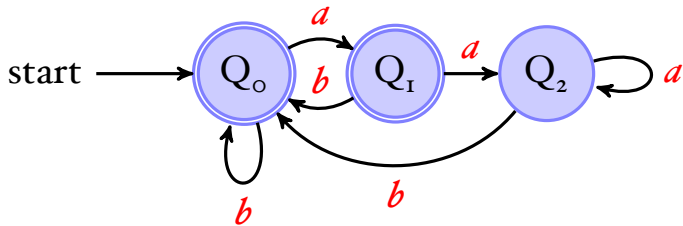
If $q =$

Finally:

$$Q_0 = (b + ab + aa(a^*)b)^*$$

$$Q_I = (b + ab + aa(a^*)b)^* a$$

$$Q_2 = (b + ab + aa(a^*)b)^* aa(a^*)$$



$$Q_0 = \mathbf{I} + Q_0 b + Q_I b + Q_2 b$$

$$Q_I = Q_0 a$$

$$Q_2 = Q_I a + Q_2 a$$

Arden's Lemma:

If $q =$

Finally:

$$Q_0 = (b + ab + aa(a^*)b)^*$$

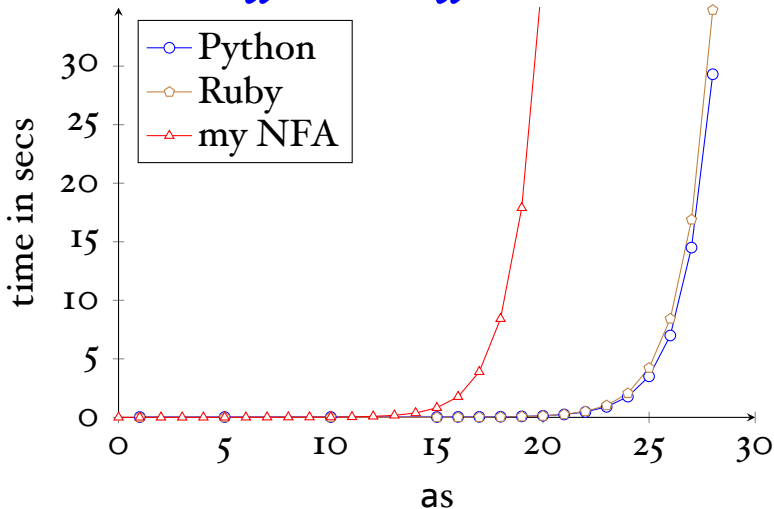
$$Q_I = (b + ab + aa(a^*)b)^* a$$

$$Q_2 = (b + ab + aa(a^*)b)^* aa(a^*)$$

Regexps and Automata



$$a^{\{n\}} \cdot a^{\{n\}}$$



The punchline is that existing libraries do depth-first search in NFAs.

Regular Languages

Two equivalent definitions:

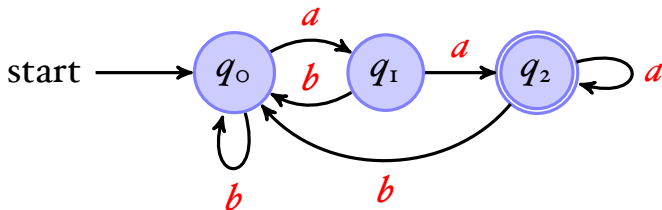
A language is **regular** iff there exists a regular expression that recognises all its strings.

A language is **regular** iff there exists an automaton that recognises all its strings.

for example $a^n b^n$ is not regular

Negation

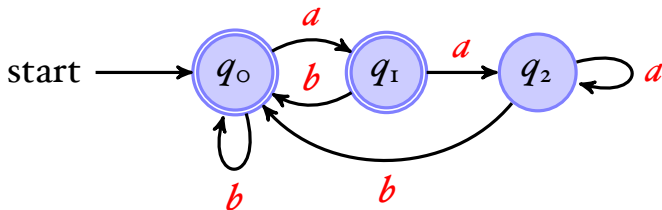
Regular languages are closed under negation:



But requires that the automaton is **completed!**

Negation

Regular languages are closed under negation:



But requires that the automaton is **completed!**

The Goal of this Course

Write a compiler



Today a lexer.

Regular Expressions

In programming languages they are often used to recognise:

- symbols, 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*.

The same problem with

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

and the string *abc*.

Or, keywords are `if` 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 Matcher

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

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

Sulzmann Matcher

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

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

Sulzmann Matcher

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



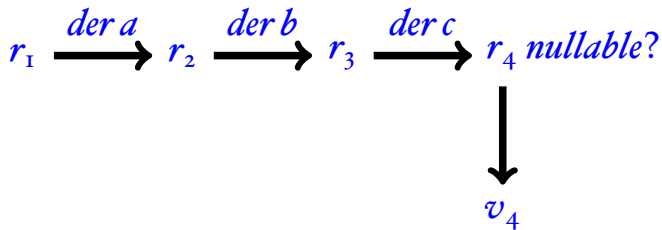
Sulzmann Matcher

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



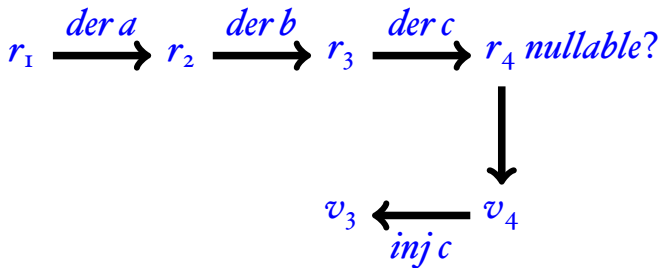
Sulzmann Matcher

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



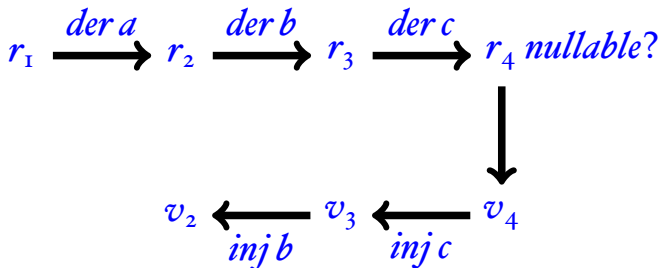
Sulzmann Matcher

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



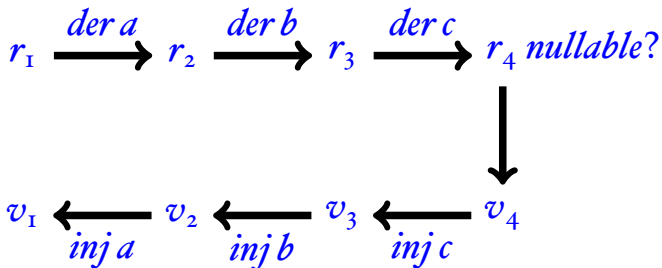
Sulzmann Matcher

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



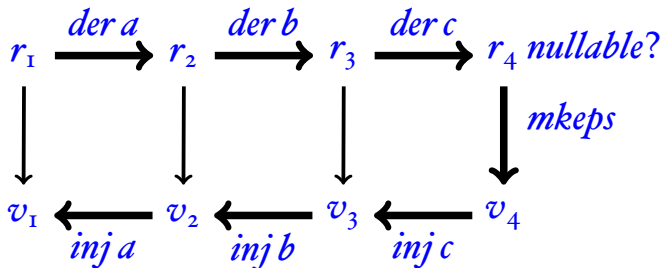
Sulzmann Matcher

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



Sulzmann Matcher

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



Regexes and Values

Regular expressions and their corresponding values:

$r ::=$	\emptyset	$v ::=$	<i>Empty</i>
	\mathbf{I}		<i>Char</i> (c)
	c		<i>Seq</i> (v_1, v_2)
	$r_1 \cdot r_2$		<i>Left</i> (v)
	$r_1 + r_2$		<i>Right</i> (v)
	r^*		<i>Stars</i> []
			<i>Stars</i> [v_1, \dots, v_n]

```
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
```

Mkeps

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

$$mkeps(\mathbf{I}) \stackrel{\text{def}}{=} \textit{Empty}$$

$$mkeps(r_1 + r_2) \stackrel{\text{def}}{=} \begin{array}{l} \textit{if nullable}(r_1) \\ \textit{then Left}(mkeps(r_1)) \\ \textit{else Right}(mkeps(r_2)) \end{array}$$

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

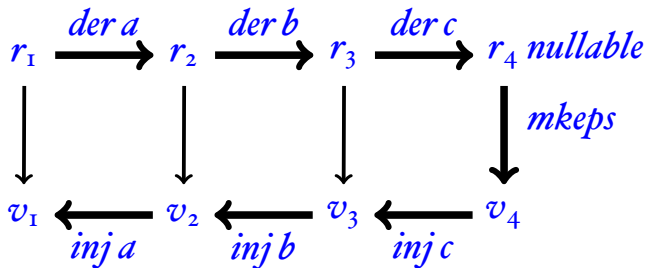
$$mkeps(r^*) \stackrel{\text{def}}{=} \textit{Stars} \ []$$

Inject

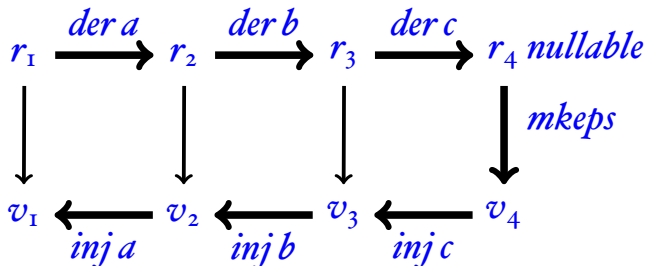
Injecting (“Adding”) a character to a value

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

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



$$\begin{aligned}
 r_1: & a \cdot (b \cdot c) \\
 r_2: & \mathbf{I} \cdot (b \cdot c) \\
 r_3: & (\mathbf{O} \cdot (b \cdot c)) + (\mathbf{I} \cdot c) \\
 r_4: & (\mathbf{O} \cdot (b \cdot c)) + ((\mathbf{O} \cdot c) + \mathbf{I})
 \end{aligned}$$

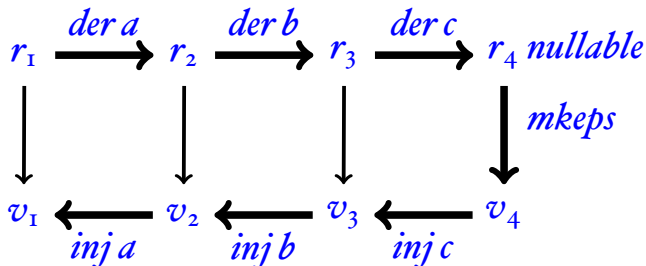


$$\begin{aligned}
 v_1: & \text{Seq}(\text{Char}(a), \text{Seq}(\text{Char}(b), \text{Char}(c))) \\
 v_2: & \text{Seq}(\text{Empty}, \text{Seq}(\text{Char}(b), \text{Char}(c))) \\
 v_3: & \text{Right}(\text{Seq}(\text{Empty}, \text{Char}(c))) \\
 v_4: & \text{Right}(\text{Right}(\text{Empty}))
 \end{aligned}$$

Flatten

Obtaining the string underlying a value:

$ 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 @ v_2 $
$ [v_1, \dots, v_n] $	$\stackrel{\text{def}}{=}$	$ v_1 @ \dots @ v_n $

$$\begin{aligned}
 r_1: & a \cdot (b \cdot c) \\
 r_2: & \mathbf{I} \cdot (b \cdot c) \\
 r_3: & (\mathbf{O} \cdot (b \cdot c)) + (\mathbf{I} \cdot c) \\
 r_4: & (\mathbf{O} \cdot (b \cdot c)) + ((\mathbf{O} \cdot c) + \mathbf{I})
 \end{aligned}$$


$$\begin{aligned}
 v_1: & \text{Seq}(\text{Char}(a), \text{Seq}(\text{Char}(b), \text{Char}(c))) \\
 v_2: & \text{Seq}(\text{Empty}, \text{Seq}(\text{Char}(b), \text{Char}(c))) \\
 v_3: & \text{Right}(\text{Seq}(\text{Empty}, \text{Char}(c))) \\
 v_4: & \text{Right}(\text{Right}(\text{Empty}))
 \end{aligned}$$

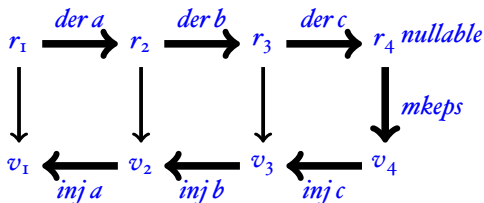
$$\begin{aligned}
 |v_1|: & abc \\
 |v_2|: & bc \\
 |v_3|: & c \\
 |v_4|: & []
 \end{aligned}$$

Lexing

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

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

lex: returns a value



Records

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

Records

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

Records

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

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

- A regular expression for email addresses

(name: $[a-z0-9_.-]^+$).@.
(domain: $[a-z0-9.-]^+$) ..
(top_level: $[a-z.]{2,6}$)

christian.urban@kcl.ac.uk

- the result environment:

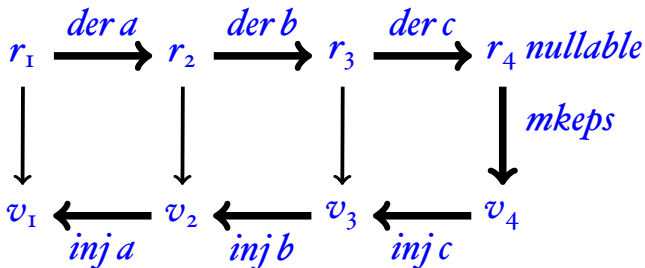
$[(name : christian.urban),$
 $(domain : kcl),$
 $(top_level : ac.uk)]$

While Tokens

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



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

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:

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

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

$$r \cdot \mathbf{I} \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 v, f_2 \text{Empty})$$

$$\mathbf{I} \cdot r \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 \text{Empty}, f_2 v)$$

$$r + \mathbf{0} \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

$$\mathbf{0} + r \mapsto r \quad \lambda f_1 f_2 v. \text{Right}(f_2 v)$$

$$r + r \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

Rectification

rectification
functions:

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

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

$$r \cdot \mathbf{I} \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 v, f_2 \text{Empty})$$

$$\mathbf{I} \cdot r \mapsto r \quad \lambda f_1 f_2 v. \text{Seq}(f_1 \text{Empty}, f_2 v)$$

$$r + \mathbf{0} \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

$$\mathbf{0} + r \mapsto r \quad \lambda f_1 f_2 v. \text{Right}(f_2 v)$$

$$r + r \mapsto r \quad \lambda f_1 f_2 v. \text{Left}(f_1 v)$$

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

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

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

$\text{case } v = Right(v') : \text{return } Right(f_2(v'))$

```

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

$simp(r): \dots$

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

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

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

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

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

```

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_Void1(f1s, f2s))
      case (_, ONE) => (r1s, F_SEQ_Void2(f1s, f2s))
      case _ => (SEQ(r1s, r2s), F_SEQ(f1s, f2s))
    }
  }
}
...

```

```

def F_SEQ_Void1(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(Void), f2(v))

```

```

def F_SEQ_Void2(f1: Val => Val, f2: Val => Val) =
  (v:Val) => Sequ(f1(v), f2(Void))

```

```

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

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

Rectification Example

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

$$f_{s_1} = \lambda v.v$$

$$f_{s_2} = \lambda v.Right(v)$$

Rectification Example

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

$$\begin{aligned} f_{s_1} &= \lambda v.v \\ f_{s_2} &= \lambda v.Right(v) \end{aligned}$$

$$f_{alt}(f_{s_1}, f_{s_2}) \stackrel{\text{def}}{=} \lambda v. \text{ case } v = Left(v'): \text{ return } Left(f_{s_1}(v')) \\ \text{ case } v = Right(v'): \text{ return } Right(f_{s_2}(v'))$$

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

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

$$f_{s1} = \lambda v.v$$

$$f_{s2} = \lambda v.Right(v)$$

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

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

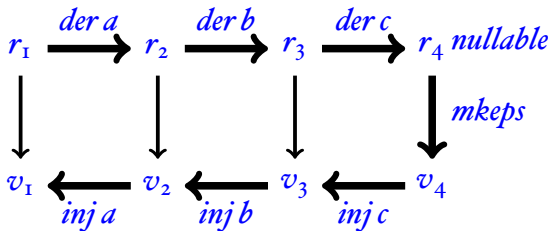
mkeps simplified case: $Right(Empty)$

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

Lexing with Simplification

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

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



Environments

Obtaining the “recorded” parts of a value:

$env(Empty)$	$\stackrel{\text{def}}{=} []$
$env(Char(c))$	$\stackrel{\text{def}}{=} []$
$env(Left(v))$	$\stackrel{\text{def}}{=} env(v)$
$env(Right(v))$	$\stackrel{\text{def}}{=} env(v)$
$env(Seq(v_1, v_2))$	$\stackrel{\text{def}}{=} env(v_1) @ env(v_2)$
$env(Stars [v_1, \dots, v_n])$	$\stackrel{\text{def}}{=} env(v_1) @ \dots @ env(v_n)$
$env(Rec(x : v))$	$\stackrel{\text{def}}{=} (x : v) :: env(v)$

While Tokens

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

”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.

$$\begin{aligned}
\text{zeroable}(\mathbf{0}) &\stackrel{\text{def}}{=} \text{true} \\
\text{zeroable}(\mathbf{1}) &\stackrel{\text{def}}{=} \text{false} \\
\text{zeroable}(c) &\stackrel{\text{def}}{=} \text{false} \\
\text{zeroable}(r_1 + r_2) &\stackrel{\text{def}}{=} \text{zeroable}(r_1) \wedge \text{zeroable}(r_2) \\
\text{zeroable}(r_1 \cdot r_2) &\stackrel{\text{def}}{=} \text{zeroable}(r_1) \vee \text{zeroable}(r_2) \\
\text{zeroable}(r^*) &\stackrel{\text{def}}{=} \text{false}
\end{aligned}$$

$\text{zeroable}(r)$ if and only if $L(r) = \{\}$