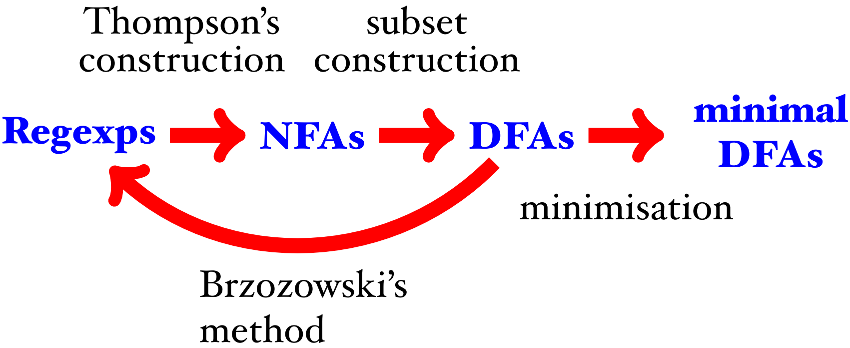
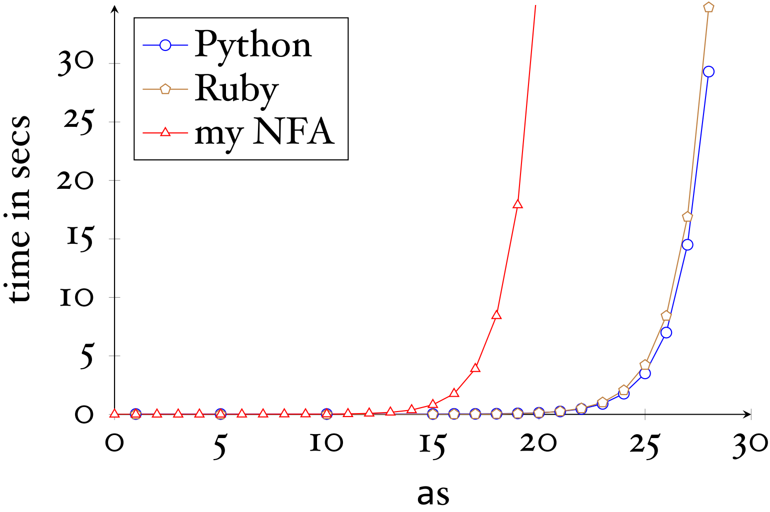
Compilers and Formal Languages (4)

Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)
Slides: KEATS (also home work is there)

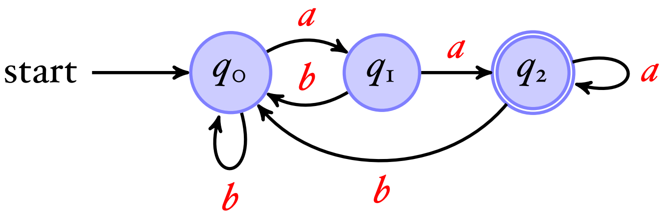
Regexps and Automata



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



DFA to Rexp



$$
\begin{aligned}
q_0 &= 1 + q_0\,b + q_1\,b + q_2\,b \qquad \text{(start state)}\\
q_1 &= q_0\,a\\
q_2 &= q_1\,a + q_2\,a
\end{aligned}
$$

Arden's Lemma:

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

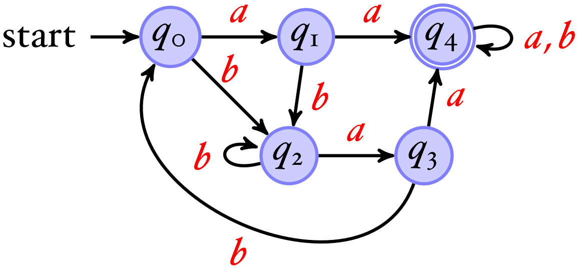
DFA Minimisation

1. Take all pairs (q, p) with $q \neq p$
2. Mark all pairs that accepting and non-accepting states
3. For all unmarked pairs (q, p) and all characters c test whether

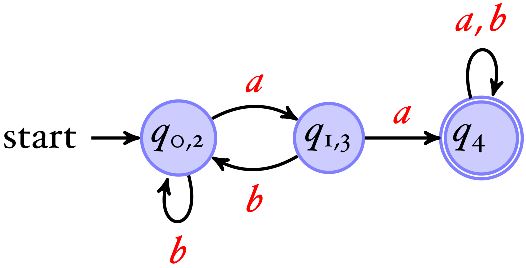
$$(\delta(q, c), \delta(p, c))$$

are marked. If yes, then also mark (q, p).

4. Repeat last step until no change.
5. All unmarked pairs can be merged.



q_1	★			
q_2		★		
q_3	★		★	
q_4	★	★	★	★
	q_0	q_1	q_2	q_3



minimal automaton

Regular Languages

Two equivalent definitions:

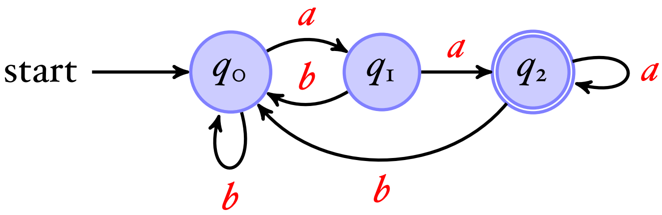
A language is regular iff there exists a regular expression that recognises all its strings.

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

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

Negation

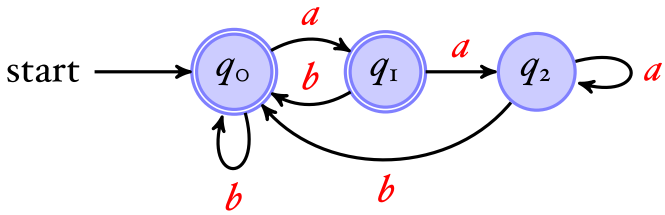
Regular languages are closed under negation:



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.

Survey: Thanks!

- **My Voice** “lecturer speaks in a low voice and is hard to hear him” “please use mic” “please use mic & lecture recording”
- **Pace** “faster pace” “a bit quick for me personally”
- **Recording** “please use recording class”
- **Module Name** “misleading”
- **Examples** “more examples”
- **Assessment** “really appreciate extension of first coursework”

Lexing

```
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
```

??

```
write "Input a number ";
read n;
while n > 1 do {
  if n % 2 == 0
  then n := n/2
  else n := 3*n+1;
};
write "Yes";
```

"if true then then 42 else +"

KEYWORD:

  if, then, else,

WHITESPACE:

  " ", \n,

IDENT:

  $\texttt{LETTER} \cdot (\texttt{LETTER} + \texttt{DIGIT} + \_)^*$

NUM:

  $(\texttt{NONZERODIGIT} \cdot \texttt{DIGIT}^*) + 0$

OP:

  +

COMMENT:

  $/* \cdot \sim(\texttt{ALL}^* \cdot (*/) \cdot \texttt{ALL}^*) \cdot */$

”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.

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 + _"*

$$iffoo$$

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:

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

Sulzmann Matcher

We want to match the string abc using r_1:

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

Sulzmann Matcher

We want to match the string abc using r_1:

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

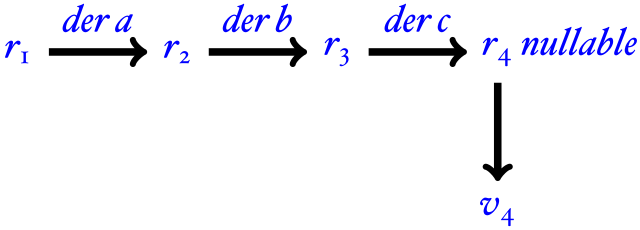
Sulzmann Matcher

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

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

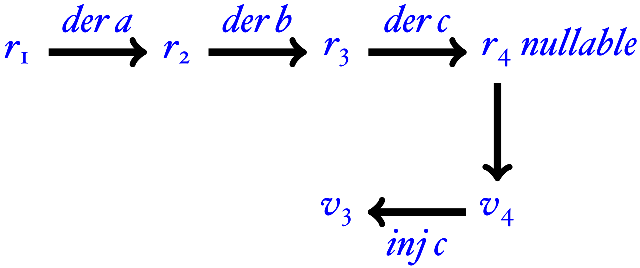
Sulzmann Matcher

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



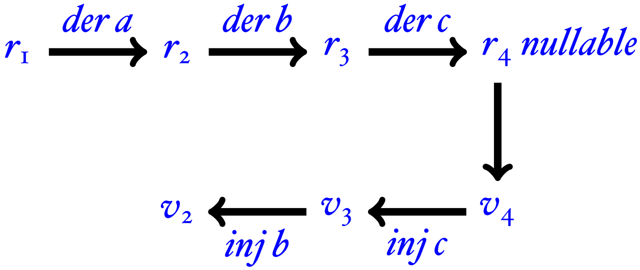
Sulzmann Matcher

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



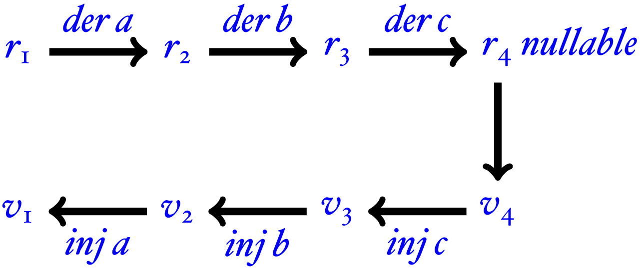
Sulzmann Matcher

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



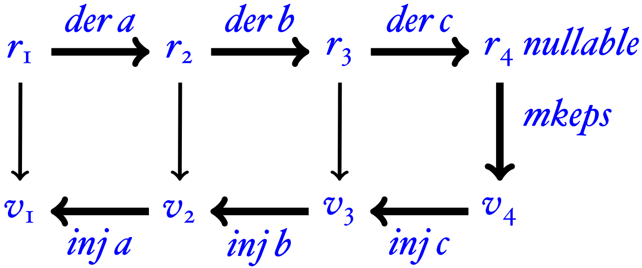
Sulzmann Matcher

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



Sulzmann Matcher

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



Regexes and Values

Regular expressions and their corresponding values:

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

```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 Seq(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:

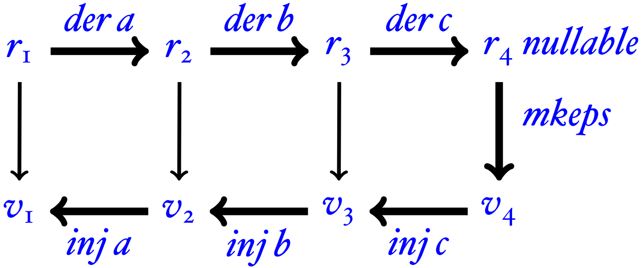
$$\begin{array}{lcl}
mkeps\, \mathbf{1} & \stackrel{\text{def}}{=} & Empty \\
mkeps\, r_1 + r_2 & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } Left(mkeps(r_1)) \\
 & & \text{else } Right(mkeps(r_2)) \\
mkeps\, r_1 \cdot r_2 & \stackrel{\text{def}}{=} & Seq(mkeps(r_1), mkeps(r_2))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

Inject

Injecting ("Adding") a character to a value

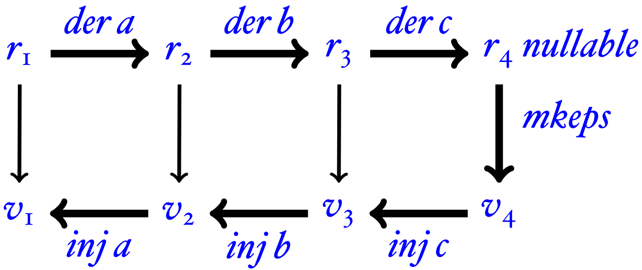
$$
\begin{array}{lcl}
inj\,(c)\,c\,Empty & \stackrel{\text{def}}{=} & Char\,c\\
inj\,(r_1 + r_2)\,c\,Left(v) & \stackrel{\text{def}}{=} & Left(inj\,r_1\,c\,v)\\
inj\,(r_1 + r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} & Right(inj\,r_2\,c\,v)\\
inj\,(r_1 \cdot r_2)\,c\,Seq(v_1, v_2) & \stackrel{\text{def}}{=}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

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



r_1
der a
r_2
der b
r_3
der c
r_4 nullable
mkeps
v_1
v_2
v_3
v_4
inj a
inj b
inj c

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:

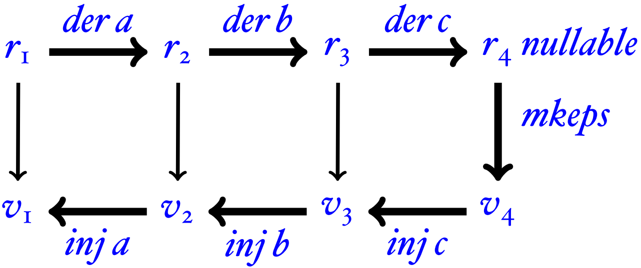
$$
\begin{array}{lcl}
|Empty| & \stackrel{\text{def}}{=} & [] \\
|Char(c)| & \stackrel{\text{def}}{=} & [c] \\
|Left(v)| & \stackrel{\text{def}}{=} & |v| \\
|Right(v)| & \stackrel{\text{def}}{=} & |v| \\
|Seq(v_1, v_2)| & \stackrel{\text{def}}{=} & |v_1| \mathbin{@} |v_2| \\
|[v_1, \ldots, v_n]| & \stackrel{\text{def}}{=}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

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

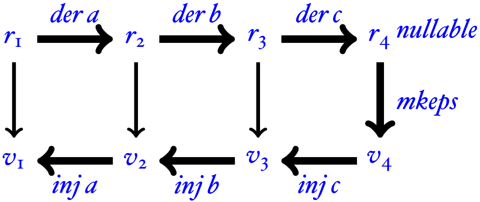
$|v_3|$: c

$|v_4|$: $[]$

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

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

`christian.urban@kcl.ac.uk`

- the result environment:

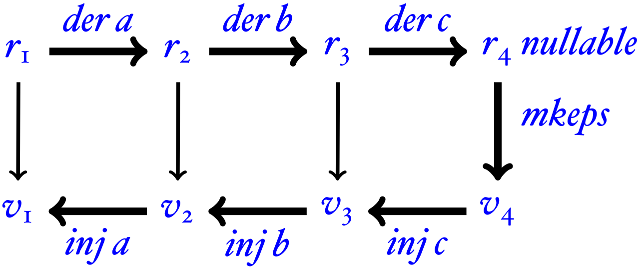
$$
\begin{array}{l}
[(\textit{name} : \texttt{christian.urban}), \\
\ (\textit{domain} : \texttt{kcl}), \\
\ (\textit{top\_level} : \texttt{ac.uk})]
\end{array}
$$

While Tokens

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

Simplification

- If we simplify after the derivative, then we are builing 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

$$Right(Right(Empty))$$

But now we simplify to $\mathbf{1}$ and produce $Empty$.

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 (NULL, _) => (r2s, F_RIGHT(f2s))
      case (_, NULL) => (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 $(r_{1s} \cdot r_{2s}, f_{seq}(f_{1s}, f_{2s}))$

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

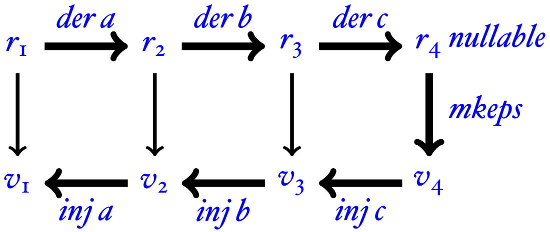
$$\lambda v.\ \text{case } v = Seq(v_1, v_2)\text{: return } Seq(f_1(v_1), f_2(v_2))$$

```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 (NULL, _) => (NULL, F_ERROR)
      case (_, NULL) => (NULL, F_ERROR)
      case (EMPTY, _) => (r2s, F_SEQ_Void1(f1s, f2s))
      case (_, EMPTY) => (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)) }
```

Lexing with Simplification

$$\begin{aligned}
lex\; r\; [] &\stackrel{\text{def}}{=} \text{if } nullable(r) \text{ then } mkeps(r) \text{ else } error\\
lex\; r\; c :: s &\stackrel{\text{def}}{=} \text{let } (r', frect) = simp(der(c, r))\\
&\qquad inj\; r\; c\; (frect(lex(r', s)))
\end{aligned}$$



$$
\begin{array}{lcl}
zeroable(\varnothing) & \stackrel{\text{def}}{=} & true \\
zeroable(\epsilon) & \stackrel{\text{def}}{=} & false \\
zeroable(c) & \stackrel{\text{def}}{=} & false \\
zeroable(r_1 + r_2) & \stackrel{\text{def}}{=} & zeroable(r_1) \wedge zeroable(r_2) \\
zeroable(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & zeroable(r_1) \vee zeroable(r_2) \\
zeroable(r^*) & \stackrel{\text{def}}{=} & false
\end{array}
$$

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