

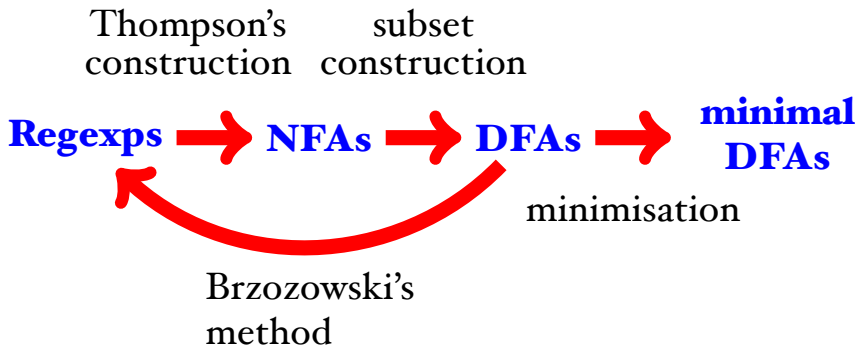
Compilers and Formal Languages (4)

Email: christian.urban at kcl.ac.uk

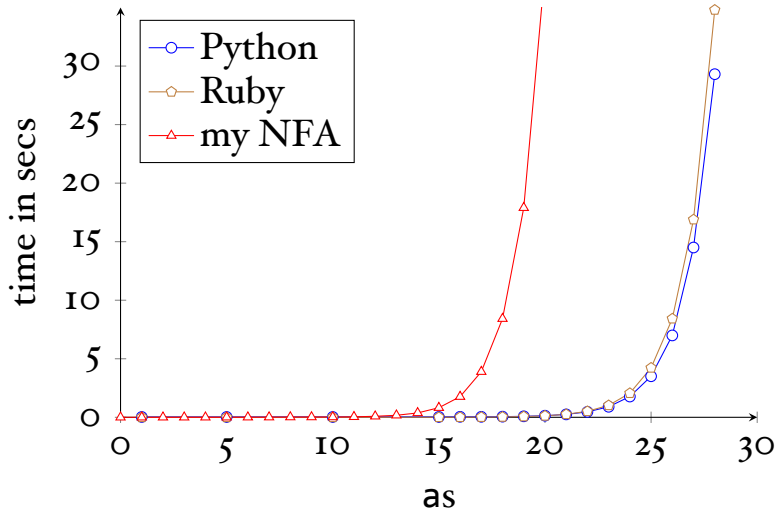
Office: SI.27 (1st floor Strand Building)

Slides: KEATS (also home work is there)

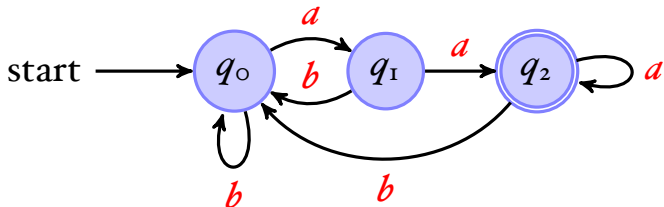
Regexps and Automata



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



DFA to Rexp



$$q_0 = \mathbf{I} + q_0 b + q_1 b + q_2 b \quad (\text{start state})$$

$$q_1 = q_0 a$$

$$q_2 = q_1 a + q_2 a$$

Arden's Lemma:

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

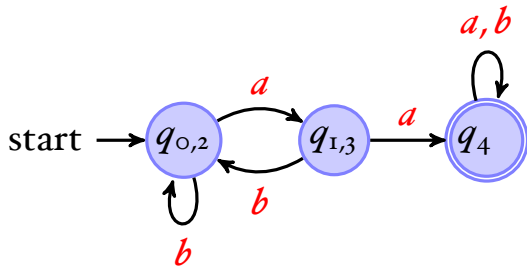
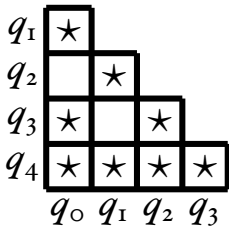
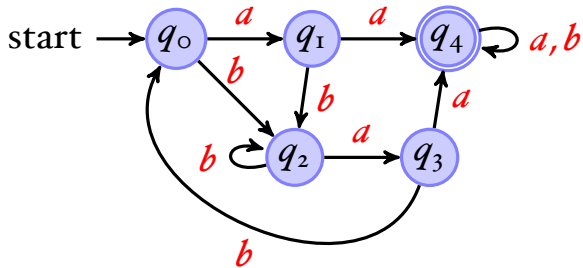
DFA Minimisation

- 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.



minimal automaton

Regular Languages

Two equivalent definitions:

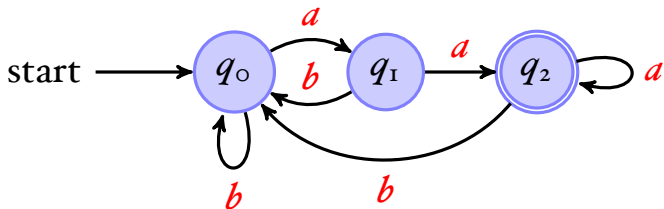
A language is **regular** iff there exists a regular expression that recognises all its strings.

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

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

Negation

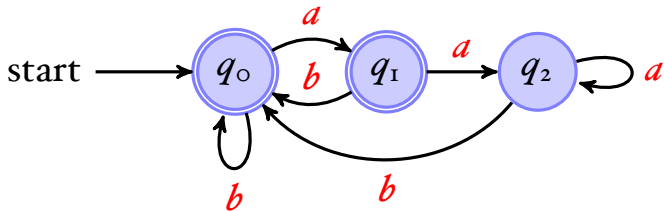
Regular languages are closed under negation:



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

```
1 write "Fib";
2 read n;
3 minus1 := 0;
4 minus2 := 1;
5 while n > 0 do {
6     temp := minus2;
7     minus2 := minus1 + minus2;
8     minus1 := temp;
9     n := n - 1
10 };
11 write "Result";
12 write minus2
```



```
1 write "Input a number ";
2 read n;
3 while n > 1 do {
4     if n % 2 == 0
5     then n := n/2
6     else n := 3*n+1;
7 };
8 write "Yes";
```

”if true then then 42 else +”

KEYWORD:

if, then, else,

WHITESPACE:

” ”, \n,

IDENT:

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 + `_`”*

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 :



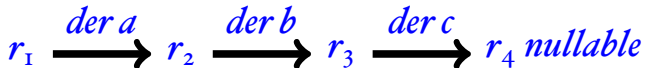
Sulzmann Matcher

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



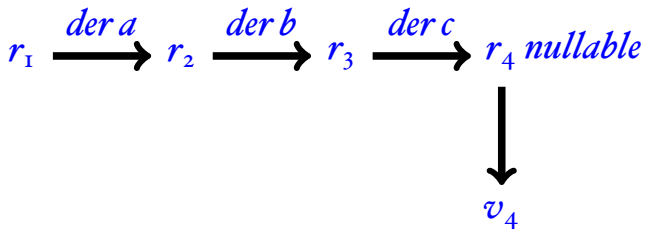
Sulzmann Matcher

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



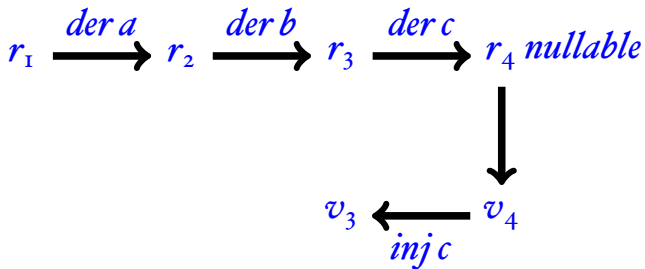
Sulzmann Matcher

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



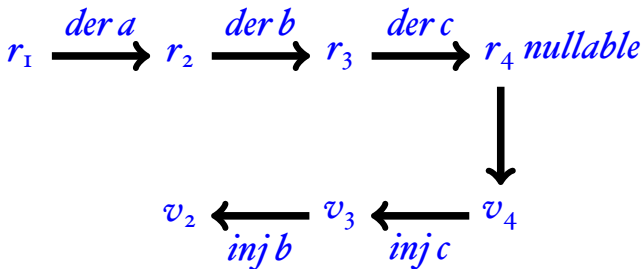
Sulzmann Matcher

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



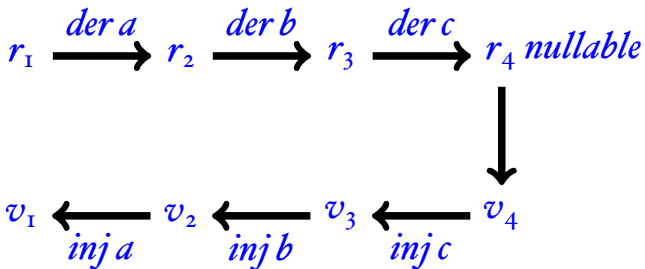
Sulzmann Matcher

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



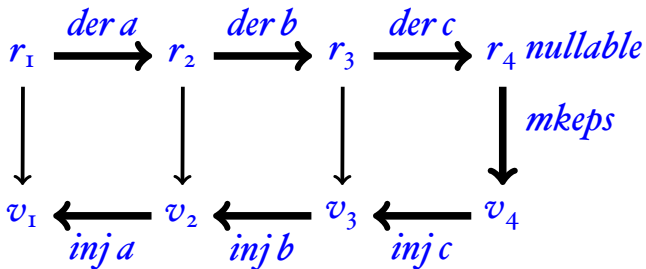
Sulzmann Matcher

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:

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

```
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 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:

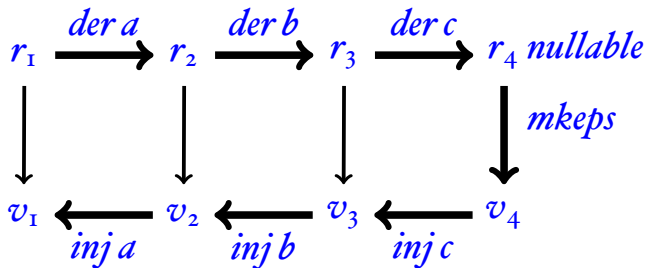
$$\begin{aligned} \mathit{mkeps} \mathbf{I} &\stackrel{\text{def}}{=} \mathit{Empty} \\ \mathit{mkeps} r_1 + r_2 &\stackrel{\text{def}}{=} \text{if } \mathit{nullable}(r_1) \\ &\quad \text{then } \mathit{Left}(\mathit{mkeps}(r_1)) \\ &\quad \text{else } \mathit{Right}(\mathit{mkeps}(r_2)) \\ \mathit{mkeps} r_1 \cdot r_2 &\stackrel{\text{def}}{=} \mathit{Seq}(\mathit{mkeps}(r_1), \mathit{mkeps}(r_2)) \\ \mathit{mkeps} r^* &\stackrel{\text{def}}{=} \square \end{aligned}$$

Inject

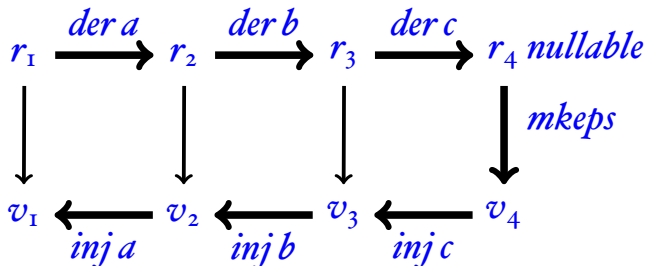
Injecting (“Adding”) a character to a value

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

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



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

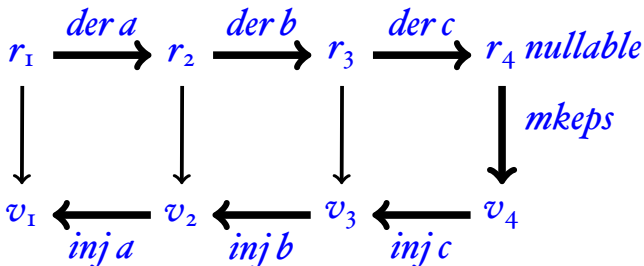


$$\begin{aligned}
 v_1: & \text{Seq}(\text{Char}(a), \text{Seq}(\text{Char}(b), \text{Char}(c))) \\
 v_2: & \text{Seq}(\text{Empty}, \text{Seq}(\text{Char}(b), \text{Char}(c))) \\
 v_3: & \text{Right}(\text{Seq}(\text{Empty}, \text{Char}(c))) \\
 v_4: & \text{Right}(\text{Right}(\text{Empty}))
 \end{aligned}$$

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 $

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


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

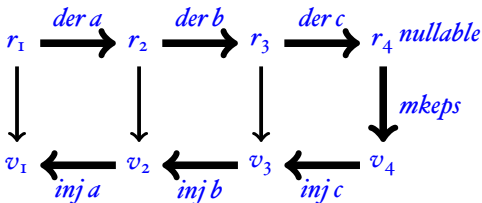
$$\begin{aligned}
 |v_1|: & abc \\
 |v_2|: & bc \\
 |v_3|: & c \\
 |v_4|: & []
 \end{aligned}$$

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:

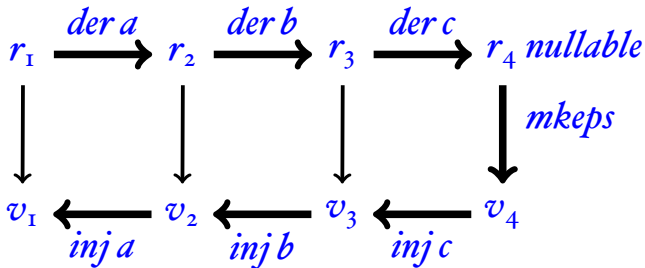
$[(name : christian.urban),$
 $(domain : kcl),$
 $(top_level : ac.uk)]$

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.



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

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 $r = r_I + r_2$

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

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

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

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

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

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

Rectification

simp(*r*):...

case $r = r_1 \cdot r_2$

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

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

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

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

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

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

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

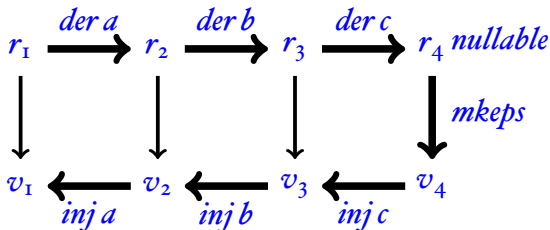
$f_{\text{seq}}(f_1, f_2) \stackrel{\text{def}}{=}$

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

Lexing with Simplification

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

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



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

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