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

Coursework

- $der\,c\,(r^+) \stackrel{\text{def}}{=} der\,c(r \cdot r^*)$ given that $r^+ \stackrel{\text{def}}{=} r \cdot r^*$

Coursework

- $der\, c\, (r^+) \stackrel{\text{def}}{=} der\, c (r \cdot r^*)$ given that $r^+ \stackrel{\text{def}}{=} r \cdot r^*$

$$der\, c\, (r \cdot r^*) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } nullable\, r \\ \text{then } (der\, c\, r) \cdot r^* + der\, c\, (r^*) \\ \text{else } (der\, c\, r) \cdot r^* \end{array}$$

Coursework

- $der\,c\,(r^+) \stackrel{\text{def}}{=} der\,c(r \cdot r^*)$ given that $r^+ \stackrel{\text{def}}{=} r \cdot r^*$

$$der\,c\,(r \cdot r^*) \stackrel{\text{def}}{=} \begin{array}{l}\text{if } nullable\,r\\ \text{then } (der\,c\,r) \cdot r^* + (der\,c\,r) \cdot r^*\\ \text{else } (der\,c\,r) \cdot r^*\end{array}$$

Coursework

- $der\,c\,(r^+) \stackrel{\text{def}}{=} der\,c(r \cdot r^*)$ given that $r^+ \stackrel{\text{def}}{=} r \cdot r^*$

$$der\,c\,(r \cdot r^*) \stackrel{\text{def}}{=} \begin{array}{l} \text{if } nullable\ r \\ \text{then } (der\,c\,r) \cdot r^* \\ \text{else } (der\,c\,r) \cdot r^* \end{array}$$

Coursework

- $der\,c\,(r^+) \stackrel{\text{def}}{=} der\,c(r \cdot r^*)$ given that $r^+ \stackrel{\text{def}}{=} r \cdot r^*$

$$der\,c\,(r \cdot r^*) \stackrel{\text{def}}{=} (der\,c\,r) \cdot r^*$$

Coursework (2)

- `CFUN(f: Char => Boolean)`

```
CHAR(c: Char) ≝
  CFUN(_ == c)

RANGE(cs: Set[Char]) ≝
  CFUN(cs.contains(_))

ALL ≝
  CFUN((c: Char) => true)
```

The Goal of this Course

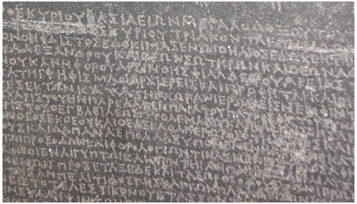
Write a compiler

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

??

"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 + _"*

$$if \qquad iffoo$$

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:

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

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:

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

Sulzmann & Lu Matcher

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

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

Sulzmann & Lu Matcher

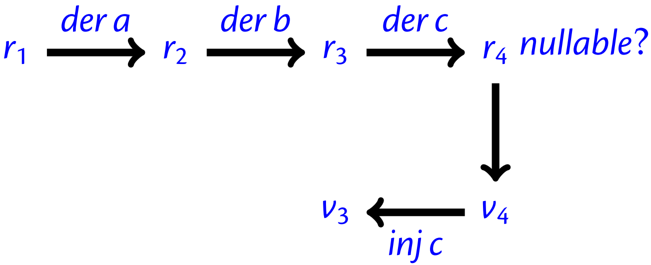
We want to match the string *abc* using r_1:

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

$$r_4 \downarrow v_4$$

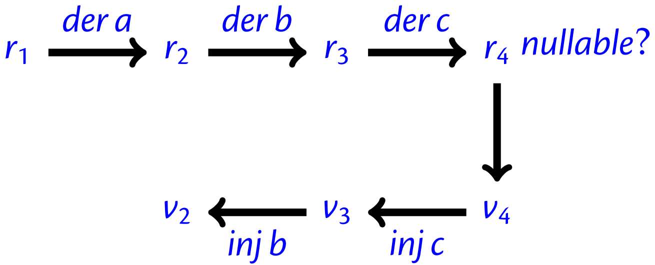
Sulzmann & Lu Matcher

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



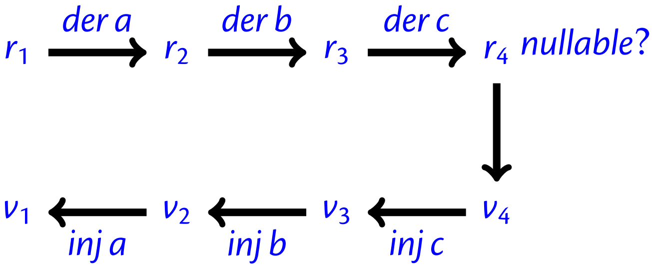
Sulzmann & Lu Matcher

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



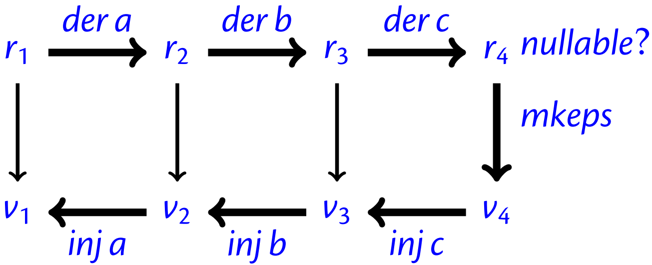
Sulzmann & Lu Matcher

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



Sulzmann & Lu Matcher

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



Regexes and Values

Regular expressions and their corresponding values:

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

```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
```

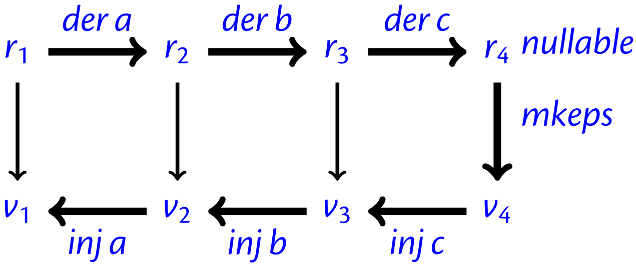
```scala
abstract class Val
case object Empty extends Val
case class Chr(c: Char) extends Val
case class Sequ(v1: Val, v2: Val) extends Val
case class Left(v: Val) extends Val
case class Right(v: Val) extends Val
case class Stars(vs: List[Val]) extends Val
```

r_1: $a \cdot (b \cdot c)$

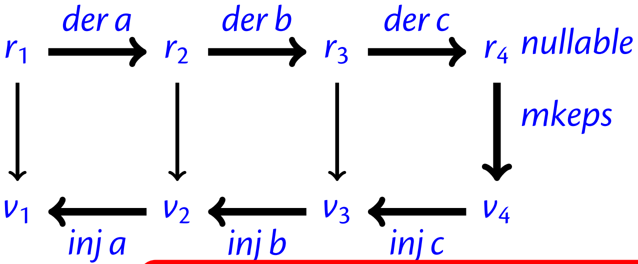
r_2: $\mathbf{1} \cdot (b \cdot c)$

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:

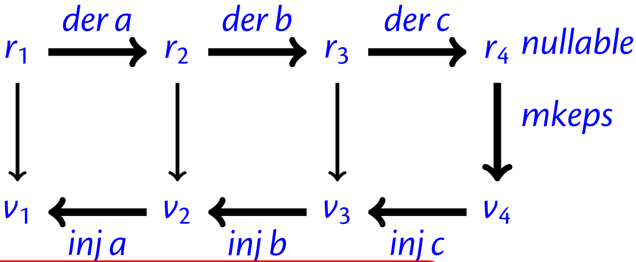
$$
\begin{array}{lcl}
|Empty| & \stackrel{\text{def}}{=} & [] \\
|Char(c)| & \stackrel{\text{def}}{=} & [c] \\
|Left(v)| & \stackrel{\text{def}}{=} & |v| \\
|Right(v)| & \stackrel{\text{def}}{=} & |v| \\
|Seq(v_1, v_2)| & \stackrel{\text{def}}{=} & |v_1| \mathbin{@} |v_2| \\
|[v_1, \ldots, v_n]| & \stackrel{\text{def}}{=}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

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

Injecting ("Adding") a character to a value

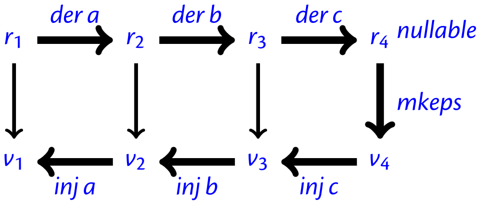
$$
\begin{array}{lcl}
inj\;(c)\;c\;(Empty) & \stackrel{\text{def}}{=} & Char\,c\\
inj\;(r_1 + r_2)\;c\;(Left(v)) & \stackrel{\text{def}}{=} & Left(inj\,r_1\,c\,v)\\
inj\;(r_1 + r_2)\;c\;(Right(v)) & \stackrel{\text{def}}{=} & Right(inj\,r_2\,c\,v)\\
inj\;(r_1 \cdot r_2)\;c\;(Seq(v_1, v_2)) & \stackrel{\text{def}}{=}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

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

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

Records

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

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

- A regular expression for email addresses

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

`christian.urban@kcl.ac.uk`

- the result environment:

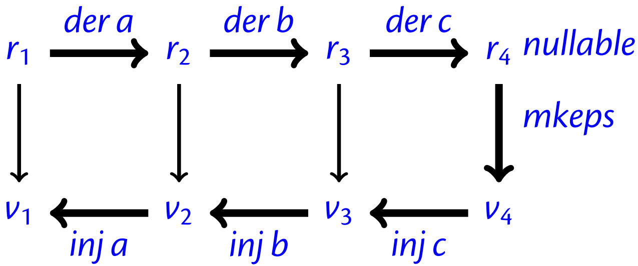
$$[(\mathit{name} : \texttt{christian.urban}), (\mathit{domain} : \texttt{kcl}), (\mathit{top\_level} : \texttt{ac.uk})]$$

While Tokens

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

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

$$Right(Right(Empty))$$

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:

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

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

$simp(r)$:...

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

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

```
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

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

Rectification Example

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

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

Rectification Example

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

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

$$f_{alt}(f_{s1}, f_{s2}) \stackrel{\text{def}}{=}$$

$$\begin{array}{ll} \lambda v. & \text{case } v = Left(v'): \quad \text{return } Left(f_{s1}(v')) \\ & \text{case } v = Right(v'): \text{return } Right(f_{s2}(v')) \end{array}$$

Rectification Example

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

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

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

Rectification Example

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

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

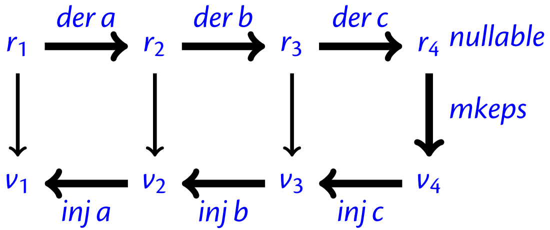
$\lambda v.$ case $v = Left(v')$: return $Left(v')$
case $v = Right(v')$: return $Right(Right(v'))$

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

Lexing with Simplification

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

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



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

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

```
"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.

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

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