

# Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk  
Office Hour: Friday 11:30 – 12:30  
Location: N7.07 (North Wing, Bush House)  
Slides & Progs: KEATS



Wifi: IET-Guest

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

# 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))\*

# The Goal of this Course

**Write a compiler**



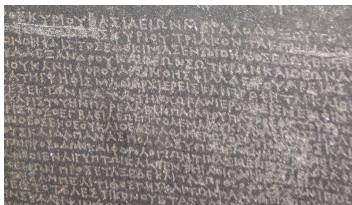
Today a lexer.

# 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](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](http://www.haskell.org/haskellwiki/Regex_Posix)

traditional lexers are fast, but hairy

# Sulzmann & Lu Lexer

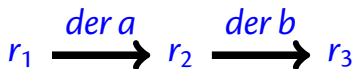
We want to match the string *abc* using  $r_1$ :

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



# Sulzmann & Lu Lexer

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



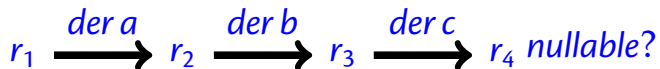
# Sulzmann & Lu Lexer

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



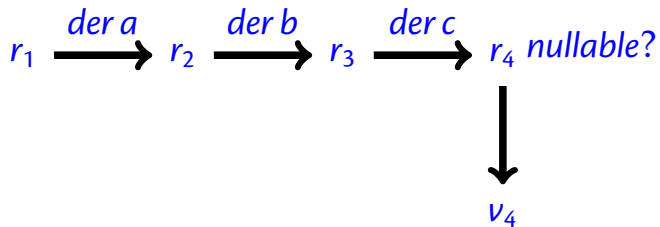
# Sulzmann & Lu Lexer

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



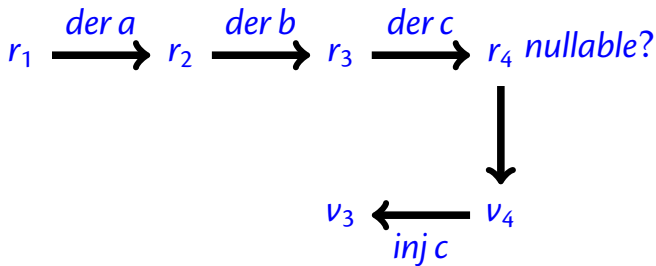
# Sulzmann & Lu Lexer

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



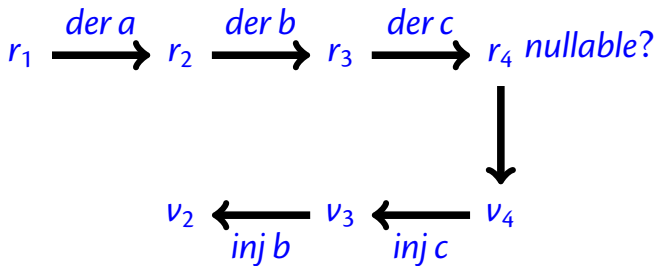
# Sulzmann & Lu Lexer

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



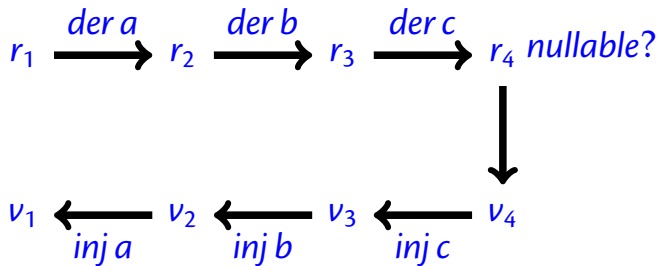
# Sulzmann & Lu Lexer

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



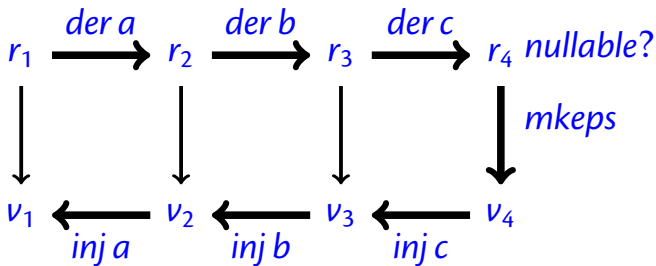
# Sulzmann & Lu Lexer

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



# Sulzmann & Lu Lexer

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





# Regexes and Values

Regular expressions and their corresponding values:

$r ::= 0$

|  $1$

|  $c$

|  $r_1 \cdot r_2$

|  $r_1 + r_2$

|  $r^*$

$v ::=$

| *Empty*

| *Char*( $c$ )

| *Seq*( $v_1, v_2$ )

| *Left*( $v$ )

| *Right*( $v$ )

| *Stars* []

| *Stars* [ $v_1, \dots v_n$ ]

```
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
```

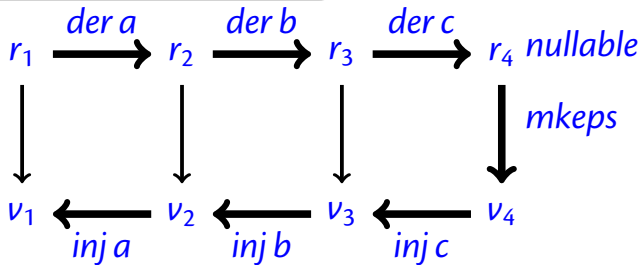
```
abstract class Val
case object Empty extends Val
case class Chr(c: Char) extends Val
case class Sequ(v1: Val, v2: Val) extends Val
case class Left(v: Val) extends Val
case class Right(v: Val) extends Val
case class Stars(vs: List[Val]) extends Val
```

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

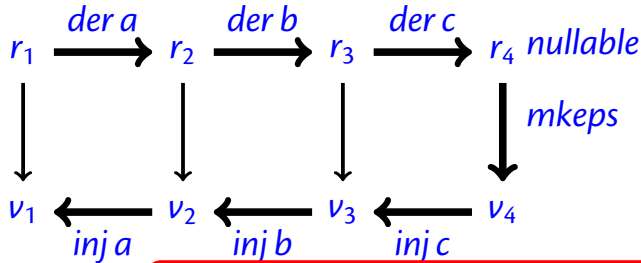
$$r_2: 1 \cdot (b \cdot c)$$

$$r_3: (0 \cdot (b \cdot c)) + (1 \cdot c)$$

$$r_4: (0 \cdot (b \cdot c)) + ((0 \cdot c) + 1)$$



$r_1: a \cdot (b \cdot c)$   
 $r_2: 1 \cdot (b \cdot c)$   
 $r_3: (0 \cdot (b \cdot c)) + (1 \cdot c)$   
 $r_4: (0 \cdot (b \cdot c)) + ((0 \cdot c) + 1)$



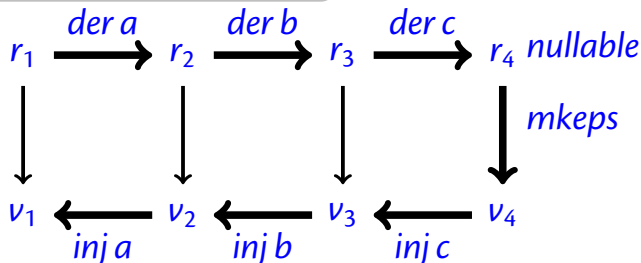
$v_1: \text{Seq}(\text{Char}(a), \text{Seq}(\text{Char}(b), \text{Char}(c)))$   
 $v_2: \text{Seq}(\text{Empty}, \text{Seq}(\text{Char}(b), \text{Char}(c)))$   
 $v_3: \text{Right}(\text{Seq}(\text{Empty}, \text{Char}(c)))$   
 $v_4: \text{Right}(\text{Right}(\text{Empty}))$

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

$r_1: a \cdot (b \cdot c)$   
 $r_2: 1 \cdot (b \cdot c)$   
 $r_3: (0 \cdot (b \cdot c)) + (1 \cdot c)$   
 $r_4: (0 \cdot (b \cdot c)) + ((0 \cdot c) + 1)$



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

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

# Mkeps

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

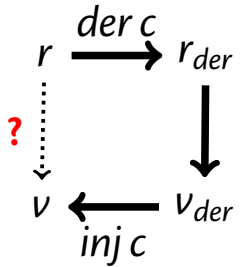
$mkeps(1) \stackrel{\text{def}}{=} Empty$

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

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

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

# 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

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

$$\begin{aligned} \text{inj } (r_1 + r_2) \text{ c } (\text{Left}(v)) &\stackrel{\text{def}}{=} \text{Left}(\text{inj } r_1 \text{ c } v) \\ \text{inj } (r_1 + r_2) \text{ c } (\text{Right}(v)) &\stackrel{\text{def}}{=} \text{Right}(\text{inj } r_2 \text{ c } v) \end{aligned}$$

$$\begin{aligned}
\text{inj } (r_1 \cdot r_2) \text{ c } (\text{Seq}(v_1, v_2)) &\stackrel{\text{def}}{=} \text{Seq}(\text{inj } r_1 \text{ c } v_1, v_2) \\
\text{inj } (r_1 \cdot r_2) \text{ c } (\text{Left}(\text{Seq}(v_1, v_2))) &\stackrel{\text{def}}{=} \text{Seq}(\text{inj } r_1 \text{ c } v_1, v_2) \\
\text{inj } (r_1 \cdot r_2) \text{ c } (\text{Right}(v)) &\stackrel{\text{def}}{=} \text{Seq}(\text{mkeps}(r_1), \text{inj } r_2 \text{ c } v)
\end{aligned}$$

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

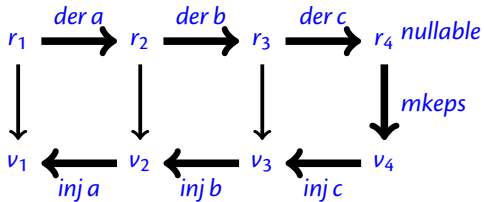
$$\text{inj}(r^*)\ c\ (\text{Seq}(v, \text{Stars}\ vs)) \stackrel{\text{def}}{=} \text{Stars}(\text{inj}\ r\ c\ v\ ::\ vs)$$

# Lexing

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

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

*lex*: returns a value



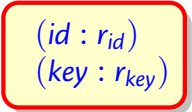
# Records

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

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

# Records

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



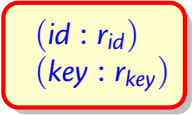
$(id : r_{id})$   
 $(key : r_{key})$



# Records

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

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



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

- A regular expression for email addresses

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

christian.urban@kcl.ac.uk

- the result environment:

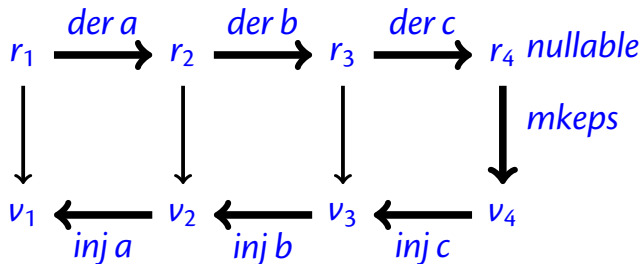
$[(name : christian.urban),$   
 $(domain : kcl),$   
 $(top\_level : ac.uk)]$

# 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))\*

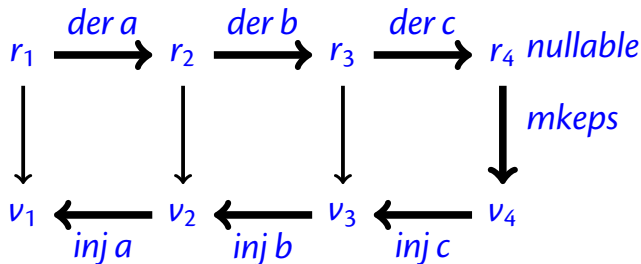
# Simplification

- If we simplify after the derivative, then we are building the value for the simplified regular expression, but **not** for the original regular expression.



# 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.



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

Normally we would have

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

and answer how this regular expression matches the empty string with the value

$$\textit{Right}(\textit{Right}(\textit{Empty}))$$

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

# Rectification

rectification

functions:

$$r \cdot 0 \mapsto 0$$

$$0 \cdot r \mapsto 0$$

$$r \cdot 1 \mapsto r \quad \lambda f_1 f_2 v. Seq(f_1 v, f_2 Empty)$$

$$1 \cdot r \mapsto r \quad \lambda f_1 f_2 v. Seq(f_1 Empty, f_2 v)$$

$$r + 0 \mapsto r \quad \lambda f_1 f_2 v. Left(f_1 v)$$

$$0 + r \mapsto r \quad \lambda f_1 f_2 v. Right(f_2 v)$$

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

# Rectification

rectification  
functions:

$$r \cdot 0 \mapsto 0$$

$$0 \cdot r \mapsto 0$$

$$r \cdot 1 \mapsto r \quad \lambda f_1 f_2 v. Seq(f_1 v, f_2 Empty)$$

$$1 \cdot r \mapsto r \quad \lambda f_1 f_2 v. Seq(f_1 Empty, f_2 v)$$

$$r + 0 \mapsto r \quad \lambda f_1 f_2 v. Left(f_1 v)$$

$$0 + r \mapsto r \quad \lambda f_1 f_2 v. Right(f_2 v)$$

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

old *simp* returns a rexp;

new *simp* returns a rexp and a rectification function.



# Rectification $\_ + \_$

$\text{simp}(r)$ :

case  $r = r_1 + r_2$

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

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

case  $r_{1s} = \mathbf{0}$ : return  $(r_{2s}, \lambda v. \text{Right}(f_{2s}(v)))$

case  $r_{2s} = \mathbf{0}$ : return  $(r_{1s}, \lambda v. \text{Left}(f_{1s}(v)))$

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

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

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

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

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

```

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 \_ · \_

$\text{simp}(r)$ :...

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

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

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

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

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

```

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_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

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

# Rectification Example

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

# Rectification Example

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

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

# Rectification Example

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

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

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



# Rectification Example

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

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

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

# Rectification Example

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

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

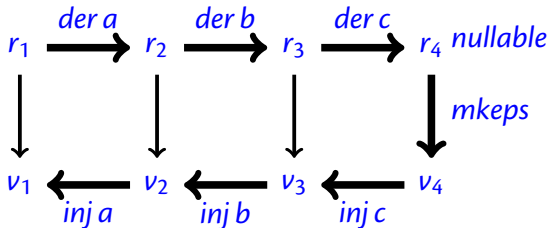
$\lambda v.$  case  $v = \text{Left}(v')$ : return  $\text{Left}(v')$   
case  $v = \text{Right}(v')$ : return  $\text{Right}(\text{Right}(v'))$

*mkeps* simplified case:  $\text{Right}(\text{Empty})$   
rectified case:  $\text{Right}(\text{Right}(\text{Empty}))$

# Lexing with Simplification

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

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



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



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





























































