Let's play with Regular Expressions

Sergey Vinokurov

serg.foo@gmail.com, @5ergv

2016-11-26

Outline

Introduction

- This talk is based on “A Play on Regular Expressions” Functional Pearl by Sebastian Fischer, Frank Huch and Thomas Wilke

Introduction

- This talk is based on “A Play on Regular Expressions” Functional Pearl by Sebastian Fischer, Frank Huch and Thomas Wilke
- Regular expressions are usually taken for granted and are implemented in C

Why reimplement regular expressions in Haskell

- Some widespread implementations, e.g. Perl, Python, Java take exponential time on some inputs - we can do better

Why reimplement regular expressions in Haskell

- Some widespread implementations, e.g. Perl, Python, Java take exponential time on some inputs - we can do better
  - Not all implementations - grep and tcl, among others, are OK

Why reimplement regular expressions in Haskell

- Some widespread implementations, e.g. Perl, Python, Java take exponential time on some inputs - we can do better
  - Not all implementations - grep and tcl, among others, are OK
- Foreign libraries are harder to integrate with pure code

Why reimplement regular expressions in Haskell

- Some widespread implementations, e.g. Perl, Python, Java take exponential time on some inputs - we can do better
  - Not all implementations - grep and tcl, among others, are OK
- Foreign libraries are harder to integrate with pure code
  - Have a trie data structure, want all entries where prefix matches a regex. Can do this naively but will have to rerun maching process for each prefix

Why reimplement regular expressions in Haskell

- Some widespread implementations, e.g. Perl, Python, Java take exponential time on some inputs - we can do better
  - Not all implementations - grep and tcl, among others, are OK
- Foreign libraries are harder to integrate with pure code
  - Have a trie data structure, want all entries where prefix matches a regex. Can do this naively but will have to rerun maching process for each prefix
  - Compiling via GHCJS and can't link to C libraries

Why reimplement regular expressions in Haskell

- Some widespread implementations, e.g. Perl, Python, Java take exponential time on some inputs - we can do better
  - Not all implementations - grep and tcl, among others, are OK
- Foreign libraries are harder to integrate with pure code
  - Have a trie data structure, want all entries where prefix matches a regex. Can do this naively but will have to rerun maching process for each prefix
  - Compiling via GHCJS and can't link to C libraries
- Can do symbolic manipulation on regular expression AST - just for fun

Regular Expressions refresher

Definitions

Regular Expression are an expression language for describing sets of strings.

Regular Expressions refresher

Definitions

Regular Expression are an expression language for describing sets of strings.

Strings - sequences of characters. a, b, c - characters, $aaaab, c$ - strings

Regular Expressions refresher

Definitions

Regular Expression are an expression language for describing sets of strings.

Strings - sequences of characters. a, b, c - characters, $aaaab, c$ - strings

Symbol ε denotes empty string

Regular Expressions refresher

Definitions

Regular Expression are an expression language for describing sets of strings.

Strings - sequences of characters. a, b, c - characters, $aaaab, c$ - strings

Symbol ε denotes empty string

String concatenation is denoted with $+\!\!+$: $aaaaaa +\!\!+ b = aaaaaab$

$$\forall x : x +\!\!+ \varepsilon = \varepsilon +\!\!+ x = x$$

Regular Expressions refresher

Definitions

Regular Expression are an expression language for describing sets of strings.

Strings - sequences of characters. a, b, c - characters, $aaaab, c$ - strings

Symbol ε denotes empty string

String concatenation is denoted with $+\!\!+$: $aaaaaa +\!\!+ b = aaaaaab$

$$\forall x : x +\!\!+ \varepsilon = \varepsilon +\!\!+ x = x$$

Alphabet - set of all characters, denoted Σ. E.g. $\Sigma = \{a, b, c\}$.

Regular Expressions refresher

Introducing Haskell ADT for regexps

Regular expressions are defined inductively

```
data Sigma = A | B | C
  deriving (Show, Eq, Ord, Enum, Bounded)

type Str a = [a]

data Regex a
  = Eps                       -- Empty regex
  | Sym a                     -- Singleton regex
  | Seq (Regex a) (Regex a)   -- Sequence
  | Alt (Regex a) (Regex a)   -- Alternatives
  | Rep (Regex a)             -- Repetition
  deriving (Show, Eq)
```

Regular Expressions refresher

Example

Consider regular expression $a^* \cdot b \cdot (c \mid \varepsilon)$

Regular Expressions refresher

Example

Consider regular expression $a^* \cdot b \cdot (c \mid \varepsilon)$

$$L(a^* \cdot b \cdot \varepsilon \mid c) = \{b, bc, ab, abc, aab, aabc, aaab, aaabc, \ldots\}$$

In Haskell,
$re = Seq\ (Rep\ (Sym\ A))\ (Seq\ (Sym\ B)\ (Alt\ Eps\ (Sym\ C)))$

Regular Expressions refresher

Example

Consider regular expression $a^* \cdot b \cdot (c \mid \varepsilon)$

$$L(a^* \cdot b \cdot \varepsilon \mid c) = \{b, bc, ab, abc, aab, aabc, aaab, aaabc, \ldots\}$$

In Haskell,
$re = Seq\ (Rep\ (Sym\ A))\ (Seq\ (Sym\ B)\ (Alt\ Eps\ (Sym\ C)))$

Or, with less parens
$re = Rep\ (Sym\ A)\ `Seq`\ Sym\ B\ `Seq`\ Alt\ Eps\ (Sym\ C)$

Regular Expressions refresher

Example

Consider regular expression $a^* \cdot b \cdot (c \mid \varepsilon)$

$$L(a^* \cdot b \cdot \varepsilon \mid c) = \{b, bc, ab, abc, aab, aabc, aaab, aaabc, \ldots\}$$

In Haskell,
$re = Seq\ (Rep\ (Sym\ A))\ (Seq\ (Sym\ B)\ (Alt\ Eps\ (Sym\ C)))$

Or, with less parens
$re = Rep\ (Sym\ A)\ `Seq`\ Sym\ B\ `Seq`\ Alt\ Eps\ (Sym\ C)$

Expected output
$accept\ re\ [B] \Rightarrow\ True$
$accept\ re\ [B, C] \Rightarrow\ True$
$accept\ re\ [A, A, A, B] \Rightarrow\ True$
$accept\ re\ [A, A, A, C] \Rightarrow\ False$
$accept\ re\ [A, A, A, B, C] \Rightarrow\ True$

Regular Expressions refresher

Defining regexp semantics

For any regex (A), notation L(A) denotes all strings that regex matches.
Let's define regexp semantics in Haskell by definning regexp matching function:

$$
\begin{array}{ll}
accept :: Eq\ a \Rightarrow Regex\ a \rightarrow Str\ a \rightarrow Bool & \\
accept\ Eps & s = acceptEps\ s \\
accept\ (Sym\ c) & s = acceptSym\ c\ s \\
accept\ (Seq\ x\ y) & s = acceptSeq\ x\ y\ s \\
accept\ (Alt\ x\ y) & s = acceptAlt\ x\ y\ s \\
accept\ (Rep\ x) & s = acceptRep\ x\ s
\end{array}
$$

Regular Expressions refresher

Empty string regex

Empty string - ε is a regular expression that matches the empty string.

$$L(\varepsilon) = \{\varepsilon\}$$

$$\begin{aligned}&acceptEps :: Str\ a \rightarrow Bool\\&acceptEps\ s = null\ s\end{aligned}$$

Regular Expressions refresher

Singleton character regex

Atoms - all symbols from Σ are regular expressions that match themselves.

$$\forall x \in \Sigma : L(x) = \{x\}$$

Haskell semantics:

$$\begin{array}{l} \mathit{acceptSym} :: \mathit{Eq}\ a \Rightarrow a \rightarrow \mathit{Str}\ a \rightarrow \mathit{Bool} \\ \mathit{acceptSym}\ c\ s = s == [c] \end{array}$$

Regular Expressions refresher

Concatenation

Concatenation - if A and B are regular expressions, so is $\mathsf{A} \cdot \mathsf{B}$

$$L(\mathsf{A} \cdot \mathsf{B}) = \{x +\!\!+\, y \mid x \in L(\mathsf{A}),\ y \in L(\mathsf{B})\}$$

Haskell semantics:

$$
\begin{array}{l}
\mathit{acceptSeq} :: \mathit{Eq}\ a \Rightarrow \mathit{Regex}\ a \rightarrow \mathit{Regex}\ a \rightarrow \mathit{Str}\ a \rightarrow \mathit{Bool} \\
\mathit{acceptSeq}\ r1\ r2\ s = \\
\quad \mathit{or}\ [\mathit{accept}\ r1\ s1\ \&\&\ \mathit{accept}\ r2\ s2 \mid (s1, s2) \leftarrow \mathit{split}\ s]
\end{array}
$$

$$
\begin{array}{l}
\mathit{split} :: \mathit{Str}\ a \rightarrow [(\mathit{Str}\ a, \mathit{Str}\ a)] \\
\mathit{split}\ xs = \mathit{map}\ (\lambda(x, y) \rightarrow (\mathit{reverse}\ x, y))\ (\mathit{go}\ [\,]\ xs) \\
\quad \mathbf{where} \\
\qquad \mathit{go}\ p\ [\,] \qquad\quad = [(p, [\,])] \\
\qquad \mathit{go}\ p\ (c : cs) = (p, c : cs) : \mathit{go}\ (c : p)\ cs
\end{array}
$$

split `"abc"` $\Rightarrow$ `[("","abc"),("a","bc"),("ab","c"),("abc","")]`

Regular Expressions refresher

Alternatives

Alternative - if A and B are regular expressions, so is A | B

$$L(\mathsf{A} \mid \mathsf{B}) = L(\mathsf{A}) \cup L(\mathsf{B})$$

Haskell semantics:

$$\textit{acceptAlt} :: \textit{Eq a} \Rightarrow \textit{Regex a} \rightarrow \textit{Regex a} \rightarrow \textit{Str a} \rightarrow \textit{Bool}$$
$$\textit{acceptAlt r1 r2 s} = \textit{accept r1 s} \mid\mid \textit{accept r2 s}$$

Regular Expressions refresher

Repetition

Repetition - also known as Kleene star, if A is a regular expression, so is A*

$$L(\mathsf{A}^*) = \{\varepsilon, \mathsf{A}, \mathsf{A}\cdot\mathsf{A}, \mathsf{A}\cdot\mathsf{A}\cdot\mathsf{A}, \ldots\}$$

Haskell semantics:

$$\begin{aligned}
&\mathit{acceptRep} :: \mathit{Eq}\ a \Rightarrow \mathit{Regex}\ a \rightarrow \mathit{Str}\ a \rightarrow \mathit{Bool}\\
&\mathit{acceptRep}\ r\ s = \mathit{or}\ [\mathit{and}\ [\mathit{accept}\ r\ p \mid p \leftarrow \mathit{ps}] \mid \mathit{ps} \leftarrow \mathit{parts}\ s]
\end{aligned}$$

$$\begin{aligned}
&\mathit{parts} :: \mathit{Str}\ a \rightarrow [[\mathit{Str}\ a]]\\
&\mathit{parts}\ [\,] \qquad = [[\,]]\\
&\mathit{parts}\ [c] \qquad = [[[c]]]\\
&\mathit{parts}\ (c : \mathit{cs}) =\\
&\quad \mathit{concat}\ [[(c : p) : \mathit{ps}, [c] : p : \mathit{ps}] \mid p : \mathit{ps} \leftarrow \mathit{parts}\ \mathit{cs}]
\end{aligned}$$

parts `"abc"` $\Rightarrow$

`[["abcd"],["a","bcd"],["ab","cd"],["a","b","cd"],["abc","d"],["a`

Regular Expressions refresher

What is not a regular expression

- Some popular implementations allow “regexps” with backreferences, like $(a|b|c)\backslash 1$

Regular Expressions refresher

What is not a regular expression

- Some popular implementations allow “regexps” with backreferences, like $(a|b|c)\backslash 1$
- $(.^*)\backslash 1$ describes non-regular language. Therefore it cannot be converted to finite automaton

Regular Expressions refresher

What is not a regular expression

- Some popular implementations allow “regexps” with backreferences, like $(a|b|c)\backslash 1$
- $(.^*)\backslash 1$ describes non-regular language. Therefore it cannot be converted to finite automaton
- Generally, we want an automaton since it's very fast

Regular Expressions refresher

What is not a regular expression

- Some popular implementations allow “regexps” with backreferences, like $(a|b|c)\backslash 1$
- $(.^*)\backslash 1$ describes non-regular language. Therefore it cannot be converted to finite automaton
- Generally, we want an automaton since it’s very fast
- We don’t include backreferences in our regexps. Regexps without them have enough expressive power

Regular Expressions refresher

Drawbacks

- Although usefull for defining semantics, *accept* function is not appropriate for practical applications.

Regular Expressions refresher

Drawbacks

- Although usefull for defining semantics, *accept* function is not appropriate for practical applications.
- Reliance on function *parts* means exponential worst-case runtime in length of string being matched

Glushkov Automaton Construction

Glushkov proposed Algorithm for generating Nondeterministic Finite Automaton, NFA, from regular expression.

NFA is a good way to run regexps because

- Size of NFA is linear in size of regular expression

Glushkov Automaton Construction

Glushkov proposed Algorithm for generating Nondeterministic Finite Automaton, NFA, from regular expression.

NFA is a good way to run regexps because

- Size of NFA is linear in size of regular expression
- Given NFA with m states we can run it on string of n characters in $O(m \cdot n)$ time

Glushkov Automaton Construction

Glushkov proposed Algorithm for generating Nondeterministic Finite Automaton, NFA, from regular expression.

NFA is a good way to run regexps because

- Size of NFA is linear in size of regular expression
- Given NFA with m states we can run it on string of n characters in $O(m \cdot n)$ time
- DFA is simpler than NFA but can have $\Omega(2^m)$ states of states for an NFA with m states.

Glushkov Automaton Construction

Example

We're going to fuse translation of regexp to NFA and execution to NFA.

Glushkov Automaton Construction

Example

We're going to fuse translation of regexp to NFA and execution to NFA.

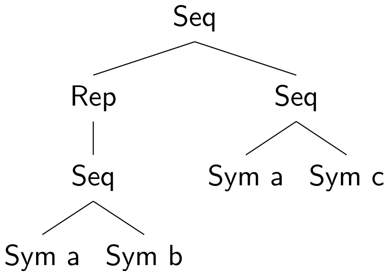
Glushkov observation: every NFA states corresponds to a position within regexp. Position is a place where a symbol occurs.

Glushkov Automaton Construction

Example

We're going to fuse translation of regexp to NFA and execution to NFA.
Glushkov observation: every NFA states corresponds to a position within regexp. Position is a place where a symbol occurs.

Let's match $(a \cdot b)^* \cdot a \cdot c$ against "ababac"



Glushkov Automaton Construction

Example - step 1

Let's match $(a \cdot b)^* \cdot a \cdot c$ against "ababac"

Glushkov Automaton Construction

Example - step 1

Let's match $(a \cdot b)^* \cdot a \cdot c$ against "ababac"

"$\boxed{\text{a}}$babac"

Glushkov Automaton Construction

Example - step 1

Let's match $(a \cdot b)^* \cdot a \cdot c$ against "ababac"

"$\boxed{\text{a}}$babac"

$$(\boxed{\text{a}} \cdot b)^* \cdot a \cdot c$$

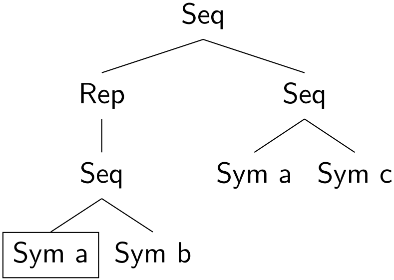
Glushkov Automaton Construction

Example - step 1

Let's match $(a \cdot b)^* \cdot a \cdot c$ against “ababac”

“$\boxed{\text{a}}$babac”

$$(\boxed{\text{a}} \cdot b)^* \cdot a \cdot c$$



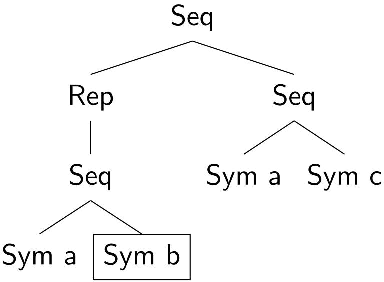
Glushkov Automaton Construction

Example - step 2

Let's match $(a \cdot b)^* \cdot a \cdot c$ against "ababac"

"a$\boxed{\text{b}}$abac"

$$(a \cdot \boxed{\text{b}})^* \cdot a \cdot c$$



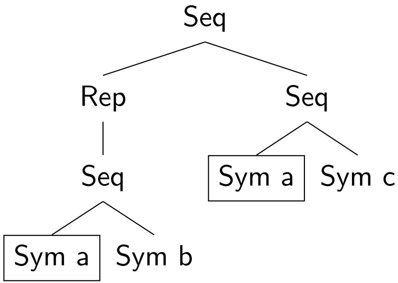
Glushkov Automaton Construction

Example - step 3

Let's match $(a \cdot b)^* \cdot a \cdot c$ against “ababac”

“ab$\boxed{\text{a}}$bac”

$$(\boxed{\text{a}} \cdot b)^* \cdot \boxed{\text{a}} \cdot c$$



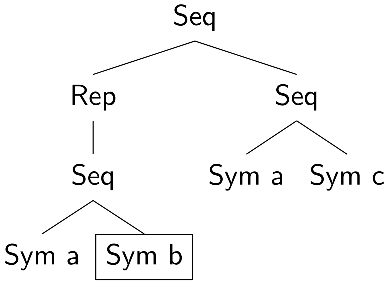
Glushkov Automaton Construction

Example - step 4

Let's match $(a \cdot b)^* \cdot a \cdot c$ against “ababac”

“aba$\boxed{\text{b}}$ac”

$$(a \cdot \boxed{\text{b}})^* \cdot a \cdot c$$



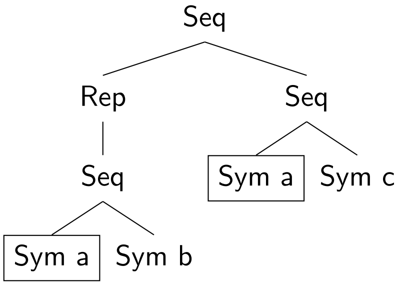
Glushkov Automaton Construction

Example - step 5

Let's match $(a \cdot b)^* \cdot a \cdot c$ against "ababac"

"abab$\boxed{\text{a}}$c"

$$(\boxed{\text{a}} \cdot b)^* \cdot \boxed{\text{a}} \cdot c$$



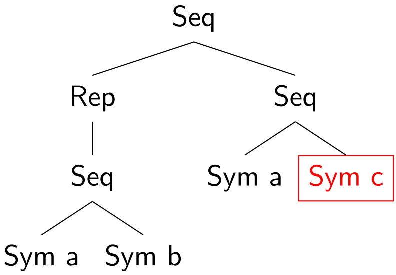
Glushkov Automaton Construction

Example - step 6

Let's match $(a \cdot b)^* \cdot a \cdot c$ against “ababac”

“ababa$\boxed{\text{c}}$”

$$(a \cdot b)^* \cdot a \cdot \boxed{\text{c}}$$



Glushkov Automaton Construction

Haskell implementation

```haskell
data Regex' a
  = Eps'                          -- Empty regex
  | Sym' Bool a                   -- Singleton regex
  | Seq' (Regex' a) (Regex' a)    -- Sequence
  | Alt' (Regex' a) (Regex' a)    -- Alternatives
  | Rep' (Regex' a)               -- Repetition
  deriving (Show, Eq)

  -- Convenient constructor
sym' :: a -> Regex' a
sym' c = Sym' False c
```

Glushkov Automaton Construction

Moving mark

$$
\begin{array}{l}
\mathit{shift} :: \mathit{Eq}\ a \Rightarrow \mathit{Bool} \to \mathit{Regex}'\ a \to a \to \mathit{Regex}'\ a \\
\mathit{shift}\ \_\ \ \mathit{Eps}'\qquad\qquad \_\ = \mathit{Eps}' \\
\mathit{shift}\ m\ (\mathit{Sym}'\ \_\ c)\ c' = \mathit{Sym}'\ (m\ \&\&\ c == c')\ c \\
\mathit{shift}\ m\ (\mathit{Alt}'\ \ p\ q)\ c\ = \mathit{Alt}'\ \ (\mathit{shift}\ m\ p\ c)\ (\mathit{shift}\ m\ q\ c) \\
\mathit{shift}\ m\ (\mathit{Seq}'\ \ p\ q)\ c\ = \\
\quad \mathit{Seq}'\ (\mathit{shift}\ m\ p\ c)\ (\mathit{shift}\ (m\ \&\&\ \mathit{empty}\ p\ ||\ \mathit{final}\ p)\ q\ c) \\
\mathit{shift}\ m\ (\mathit{Rep}'\ \ p)\quad c\ = \mathit{Rep}'\ (\mathit{shift}\ (m\ ||\ \mathit{final}\ p)\ p\ c)
\end{array}
$$

$$
\begin{array}{l}
\mathit{empty} :: \mathit{Regex}'\ a \to \mathit{Bool} \\
\mathit{empty}\ \mathit{Eps}'\qquad\quad = \mathit{True} \\
\mathit{empty}\ (\mathit{Sym}'\ \_\ \_) = \mathit{False} \\
\mathit{empty}\ (\mathit{Alt}'\ \ p\ q) = \mathit{empty}\ p\ ||\ \mathit{empty}\ q \\
\mathit{empty}\ (\mathit{Seq}'\ \ p\ q) = \mathit{empty}\ p\ \&\&\ \mathit{empty}\ q \\
\mathit{empty}\ (\mathit{Rep}'\ \ p)\quad = \mathit{True}
\end{array}
$$

Glushkov Automaton Construction

Checking if mark is in the final position

$$
\begin{aligned}
&\mathit{final} :: \mathit{Regex}'\ a \rightarrow \mathit{Bool}\\
&\mathit{final}\ \mathit{Eps}' &&= \mathit{False}\\
&\mathit{final}\ (\mathit{Sym}'\ m\ \_) &&= m\\
&\mathit{final}\ (\mathit{Alt}'\ \ p\ \ q) &&= \mathit{final}\ p\ ||\ \mathit{final}\ q\\
&\mathit{final}\ (\mathit{Seq}'\ \ p\ \ q) &&= \mathit{final}\ p\ \&\&\ \mathit{empty}\ q\ ||\ \mathit{final}\ q\\
&\mathit{final}\ (\mathit{Rep}'\ \ p) &&= \mathit{final}\ p
\end{aligned}
$$

Glushkov Automaton Construction

Matching

$$
\begin{aligned}
&\mathit{accept}' :: \mathit{Eq}\ a \Rightarrow \mathit{Regex}'\ a \rightarrow [a] \rightarrow \mathit{Bool} \\
&\mathit{accept}'\ \mathit{re}\ [\,] \qquad\quad = \mathit{empty}\ \mathit{re} \\
&\mathit{accept}'\ \mathit{re}\ (c : \mathit{cs}) = \mathit{final}\ (\mathit{foldl}\ (\mathit{shift}\ \mathit{False})\ (\mathit{shift}\ \mathit{True}\ \mathit{re}\ c)\ \mathit{cs})
\end{aligned}
$$

Glushkov Automaton Construction

Matching

$$
\begin{aligned}
&accept' :: Eq\ a \Rightarrow Regex'\ a \rightarrow [a] \rightarrow Bool\\
&accept'\ re\ [\,] \qquad\quad = empty\ re\\
&accept'\ re\ (c : cs) = final\ (foldl\ (shift\ False)\ (shift\ True\ re\ c)\ cs)
\end{aligned}
$$

Let's try our previous test regexp $a^* \cdot b \cdot (\varepsilon \mid c)$

$re = Rep'\ (sym'\ A)\ `Seq'`\ sym'\ B\ `Seq'`\ Alt'\ Eps'\ (sym'\ C)$

Expected output

$accept'\ re\ [B] \Rightarrow True$

$accept'\ re\ [B, C] \Rightarrow True$

$accept'\ re\ [A, A, A, B] \Rightarrow True$

$accept'\ re\ [A, A, A, C] \Rightarrow False$

$accept'\ re\ [A, A, A, B, C] \Rightarrow True$

Glushkov Automaton Construction

Improving efficiency

It's costly to recompute *final* and *empty* every time. We can memoize them in expressions

```
data RegexMemo a
  = EpsMemo                      -- Empty regex
  | SymMemo Bool a               -- Singleton regex
  | SeqMemo  (R a) (R a)         -- Sequence
  | AltMemo  (R a) (R a)         -- Alternatives
  | RepMemo  (R a)               -- Repetition
  deriving (Show, Eq)
data R a = R
  { rEmpty :: Bool
  , rFinal  :: Bool
  , rRegex  :: RegexMemo a
  } deriving (Show, Eq)
```

Glushkov Automaton Construction

Improving efficiency

Having defined *RegexMemo* and *R*, we can define smart constructors and we'll only need to replace all explicit constructors with smart constructors in function *shift* to get the benefit.
E.g.

$$
\begin{array}{l}
\mathit{reEps} :: R\ a \\
\mathit{reEps} = R \\
\quad \{\, \mathit{rEmpty} = \mathit{True} \\
\quad ,\mathit{rFinal} \quad = \mathit{False} \\
\quad ,\mathit{rRegex} \ = \mathit{EpsMemo} \\
\quad \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{reAlt} :: R\ a \rightarrow R\ a \rightarrow R\ a \\
\mathit{reAlt}\ p\ q = R \\
\quad \{\, \mathit{rEmpty} = \mathit{rEmpty}\ p \mid\mid \mathit{rEmpty}\ q \\
\quad ,\mathit{rFinal} \quad = \mathit{rFinal}\ p \quad \mid\mid \mathit{rFinal}\ q \\
\quad ,\mathit{rRegex} \ = \mathit{AltMemo}\ p\ q \\
\quad \}
\end{array}
$$

One cool laziness trick

Laziness allows matching strictly more expressive languages. Even some context-sensitive ones

To make our regex lazy we'll have to add another field that tracks, whether the expression was active. If it wasn't, we can avoid forcing it. This would allow us to operate infinte regexps, which can match some context-free languages and even some contex-sensitive ones.

One cool laziness trick

Laziness allows matching strictly more expressive languages. Even some context-sensitive ones

To make our regex lazy we'll have to add another field that tracks, whether the expression was active. If it wasn't, we can avoid forcing it. This would allow us to operate infinte regexps, which can match some context-free languages and even some contex-sensitive ones.

Let's match language of parentheses $\{\varepsilon, (), (()), ()(), (())(), \ldots\}$

Our alphabet is $\Sigma = \{(,)\}$

$$\textbf{data}\ \mathit{Parens} = \mathit{LParen} \mid \mathit{RParen}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq})$$

Now make regex that matches this language

$\textbf{let}\ \mathit{re} = (\mathit{symL}\ \mathit{LParen}\ \text{`}\mathit{seqL}\text{`}\ \mathit{re}\ \text{`}\mathit{seqL}\text{`}\ \mathit{symL}\ \mathit{RParen})\ \text{`}\mathit{seqL}\text{`}\ \mathit{re}\ \textbf{in}\ \mathit{re}$

One cool laziness trick

Sample run

$$\textbf{let}\ \mathit{re} = (\mathit{symL}\ \mathit{LParen}\ \text{`}\mathit{seqL}\text{`}\ \mathit{re}\ \text{`}\mathit{seqL}\text{`}\ \mathit{symL}\ \mathit{RParen})\ \text{`}\mathit{seqL}\text{`}\ \mathit{re}\ \textbf{in}\ \mathit{re}$$

One cool laziness trick

Sample run

$\textbf{let}\ \mathit{re} = (\mathit{symL}\ \mathit{LParen}\ \text{`}\mathit{seqL}\text{`}\ \mathit{re}\ \text{`}\mathit{seqL}\text{`}\ \mathit{symL}\ \mathit{RParen})\ \text{`}\mathit{seqL}\text{`}\ \mathit{re}\ \textbf{in}\ \mathit{re}$

Try it out

$\mathit{acceptLazy}\ \mathit{re}\ [\mathit{LParen}, \mathit{RParen}] \Rightarrow \mathit{True}$

$\mathit{acceptLazy}\ \mathit{re}\ [\mathit{LParen}, \mathit{RParen}, \mathit{LParen}, \mathit{RParen}] \Rightarrow \mathit{True}$

$\mathit{acceptLazy}\ \mathit{re}\ [\mathit{LParen}, \mathit{RParen}, \mathit{LParen}, \mathit{RParen}, \mathit{RParen}] \Rightarrow \mathit{False}$

Thank you!

Thank you!

Questions?