Accepting Strings
Regular Languages

• A Regular Language is a set of Strings
• Two ways to describe sets of strings $S$
  – Enumerate the strings: $S = \{s_1, s_2, s_3, \ldots\}$
  – Write a predicate – $p$: $p(x) = \text{True if } x \text{ is in the set } S$

• Problems
  – Enumeration is hard if set is infinite
  – Writing predicate varies depending upon how the set $S$ is described (RegExp, DFA, NFA, etc)
Enumeration

- Enumeration is easy to write.
- For infinite Sets, effective enumeration is only an approximation.

```haskell
meaning:: Ord a => Int -> (RegExp a) -> Set [a]
meaning n (One x) = {x}
meaning n Lambda = {""}
meaning n Empty = {}
meaning n (Union x y) = union (meaning n x) (meaning n y)
meaning n (Cat x y) = cat (meaning n x) (meaning n y)
meaning n (Star x) = starN n (meaning n x)
```
Approximating Star

\[
\begin{align*}
\text{starN } 0 \ x &= \ \{"
\}\ \\
\text{starN } 1 \ x &= \ x \\
\text{starN } n \ x &= \\
&\quad \text{union } \{"
\}\ \\
&\quad \text{(union } \ x \\
&\quad \text{(cat } \ x \\
&\quad \text{(starN } (n-1) \ x)\))
\end{align*}
\]
Approximate acceptance of RegExp

accept :: Ord a => [a] -> RegExp a -> Bool
accept s r = setElem s (meaning 3 r)
Equivalences and translation

• Since we know that DFA, NFA, NFAe, GenNFA, and RegExp all describe the same languages,
• And, we have algorithms that translate between them,
• We can translate to one and use algorithms for that one.
• Which description has the most direct acceptance algorithm?
data DFA q s =
DFA { states :: [q],
symbols :: [s],
delta :: q -> s -> q,
start :: q,
final :: [q]}

data NFA q s =
NFA { states :: [q],
symbols :: [s],
delta :: q -> s -> [q],
start :: q,
final :: [q]}

data NFAe q s =
NFAe { states :: [q],
symbols :: [s],
delta :: q -> s -> Maybe [q],
start :: q,
final :: [q]}

data RegExp a =
Lambda | Empty | One a | Union (RegExp a) (RegExp a) | Cat (RegExp a) (RegExp a) | Star (RegExp a)

data GNFA q s =
GNFA { states :: [q],
symbols :: [s],
delta :: q -> q -> RegExp s,
start :: q,
final :: [q]}

via GenNFA by RegEx decomposition
DFA Acceptance

data DFA q s = DFA { states :: [q],
    symbols :: [s],
    delta :: q -> s -> q,
    start :: q,
    final :: [q]}

trans :: (q -> s -> q) -> q -> [s] -> q
trans d q [] = q
trans d q (s:ss) = trans d (d q s) ss

accept :: (Eq q) => DFA q s -> [s] -> Bool
accept m@(DFA {delta = d, start = q0, final = f}) w = elem (trans d q0 w) f
Costs of translation

- What is the cost of translating from one specification form (RegExp, DFA, NFA, etc.) to another specification form.
Exact RegExp Acceptance

• We can write an exact RegExp acceptance function.
• It depends upon two functions of RegExp

emptyString :: RegExp sigma -> Bool
  – Can the input accept the empty string?

derivative :: RegExp s -> s -> RegExp s
  – If a RegExp can accept a string that starts with s, then what regular expression would accept everything but s?
Derivative

• if “abd...” element of the set denoted by R
• Then what regular expression R’ has the property that
• “bc...” element the set denoted by R’

• We call R’ the derivative of R with respect to ‘a’
<table>
<thead>
<tr>
<th>string</th>
<th>reg-exp</th>
<th>derivative</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;xabbc&quot;</td>
<td>$x(a+d)b^*c$</td>
<td>$(a+d)b^*c$</td>
</tr>
<tr>
<td>&quot;abbc&quot;</td>
<td>$(a+d)b^*c$</td>
<td>$b^*c$</td>
</tr>
<tr>
<td>&quot;bbc&quot;</td>
<td>$b^*c$</td>
<td>$b^*c$</td>
</tr>
<tr>
<td>&quot;bc&quot;</td>
<td>$b^*c$</td>
<td>$b^*c$</td>
</tr>
<tr>
<td>&quot;c&quot;</td>
<td>$b^*c$</td>
<td>$Λ$</td>
</tr>
</tbody>
</table>
emptyString :: RegExp a -> Bool
emptyString Lambda = True
emptyString Empty = False
emptyString (One a) = False
emptyString (Union x y) = emptyString x || emptyString y
emptyString (Star _) = True
emptyString (Cat x y) = emptyString x && emptyString y
derivative

deriv :: Ord a => RegExp a -> a -> RegExp a
deriv (One a) b | a==b = Lambda
deriv (One a) b = Empty
deriv Empty a = Empty
deriv Lambda a = Empty
deriv (Cat x y) a | not(emptyString x) = Cat (deriv x a) y
deriv (Cat x y) a =
    Union (catOpt (deriv x a) y) (deriv y a)
deriv (Union x y) a = Union (deriv x a) (deriv y a)
deriv (Star x) a = Cat (deriv x a) (Star x)
Exact Acceptance

\[
\operatorname{recog} :: [a] \to \operatorname{RegExp} a \to \text{Bool}
\]

\[
\operatorname{recog} s \operatorname{Empty} = \text{False}
\]

\[
\operatorname{recog} [] r = \operatorname{emptyString} r
\]

\[
\operatorname{recog} (x:xs) r = \operatorname{recog} xs (\operatorname{deriv} r x)
\]