Accepting Strings

Regular Languages

- A Regular Language is a set of Strings
- Two ways to describe sets of strings S
  - Enumerate the strings: S = {s1, s2, s3, …}
  - Write a predicate – p: p(x)=True if x 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.

```
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

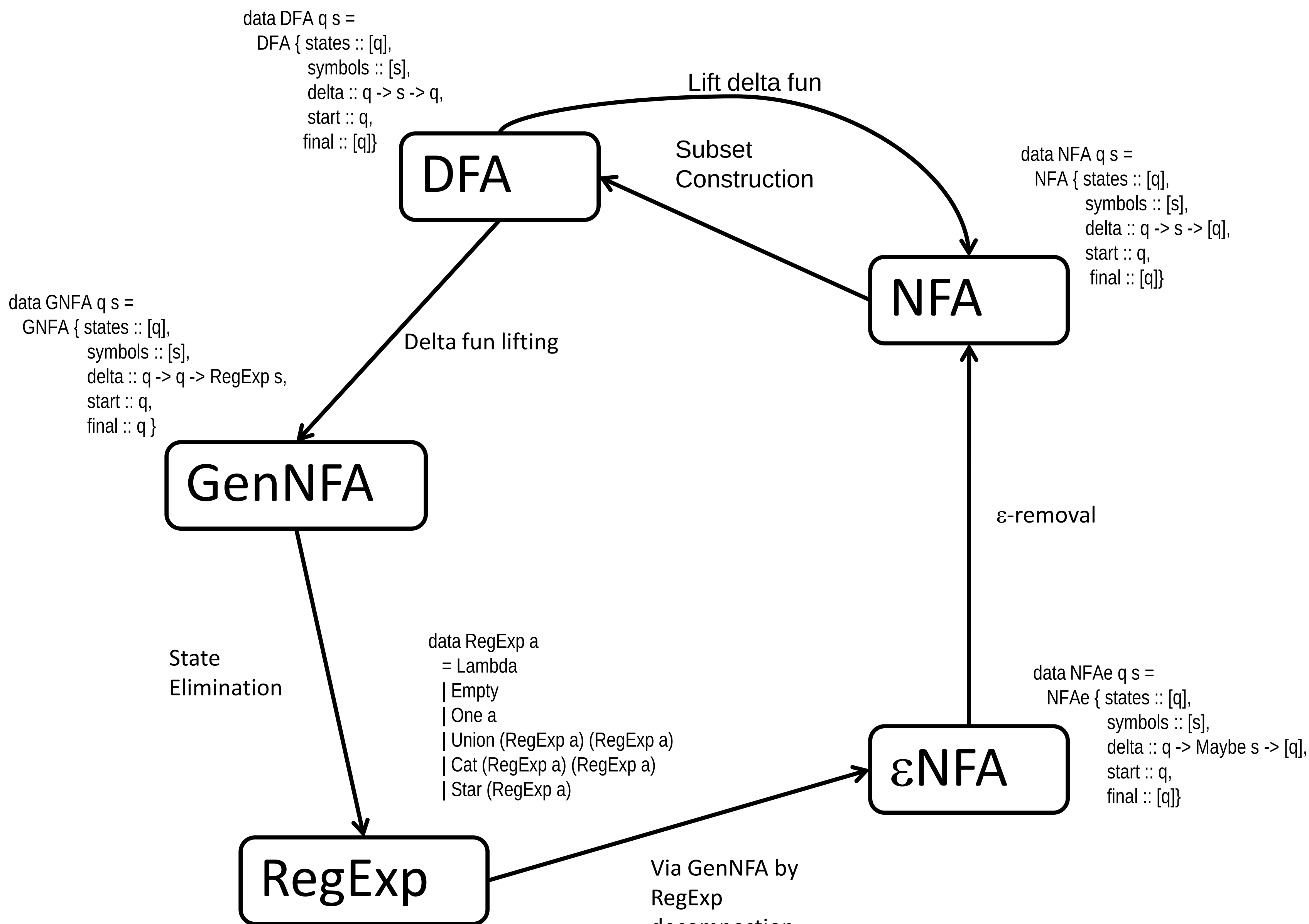
```
starN 0 x = {””}
starN 1 x = x
starN n x =
    union {””}
          (union x
                 (cat x
                      (starN (n-1) x)))
```

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]}
DFA
Lift delta fun
Subset Construction
data NFA q s =
NFA { states :: [q],
symbols :: [s],
delta :: q -> s -> [q],
start :: q,
final :: [q]}
NFA
data GNFA q s =
GNFA { states :: [q],
symbols :: [s],
delta :: q -> q -> RegExp s,
start :: q,
final :: q }
Delta fun lifting
GenNFA
ε-removal
State Elimination
data RegExp a
= Lambda
| Empty
| One a
| Union (RegExp a) (RegExp a)
| Cat (RegExp a) (RegExp a)
| Star (RegExp a)
data NFAe q s =
NFAe { states :: [q],
symbols :: [s],
delta :: q -> Maybe s -> [q],
start :: q,
final :: [q]}
εNFA
RegExp
Via GenNFA by RegExp decompostion

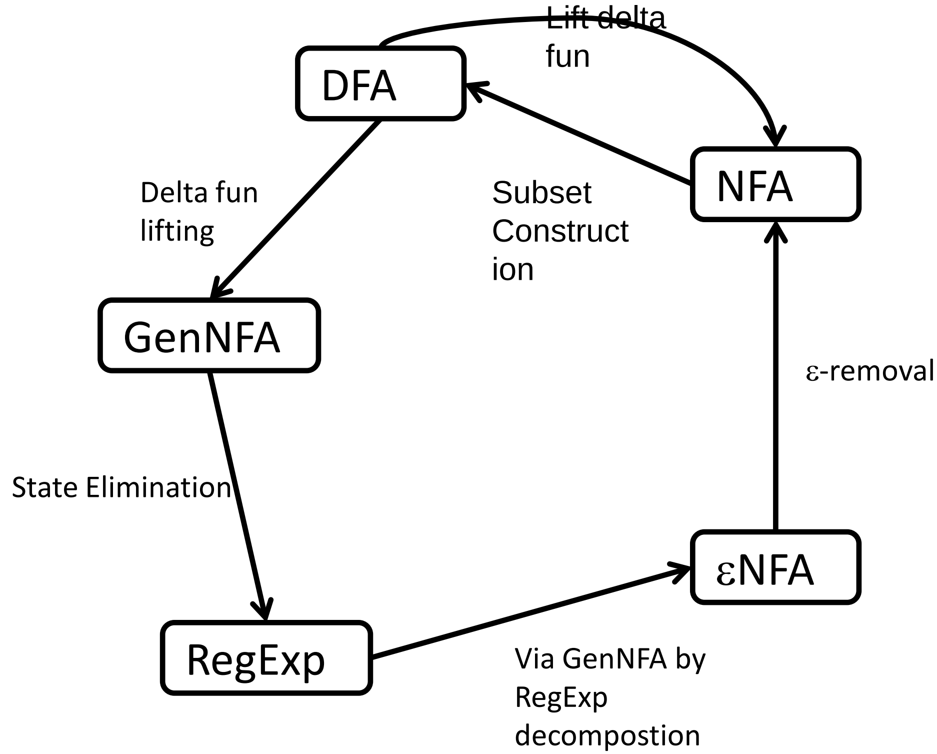
DFA Acceptance

```
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
```

This is $\underline{\delta}$

Costs of translation

- Whats the cost of translating from one specification form (RegExp, DFA, NFA, etc.) to another specification form.



Regular Expressions can be analyzed

- We saw earlier that a regular expression can be analyzed to translate it into an Λ-NFA
- Can we use a similar analysis to encode acceptance of a string by a regular expression directly, without translating into another equivalent form (DFA, NFA, etc).

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
- “bd…” element the set denoted by R’

- We call R’ the derivative of R with respect to ‘a’

string	reg-exp	derivative
"xabbc“	x(a+d)b*c	(a+d)b*c
"abbc“	(a+d)b*c	b*c
"bbc“	b*c	b*c
"bc"	b*c	b*c
"c“	b*c	Λ

emptystring

```
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 = if a==b then Lambda else Empty
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

```
recog:: [a] -> RegExp a -> Bool

recog s Empty = False
recog [] r = emptyString r
recog (x:xs) r = recog xs (deriv r x)
```