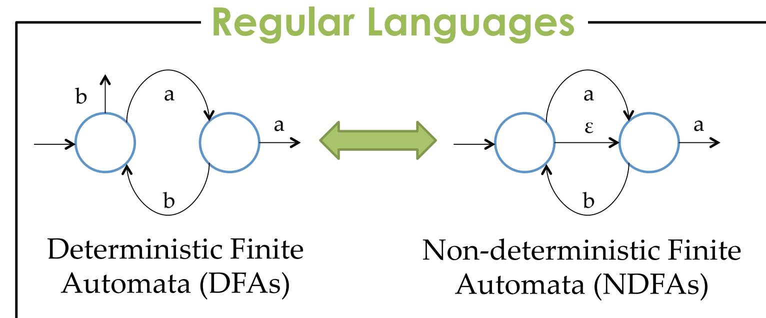
Computability and Logic

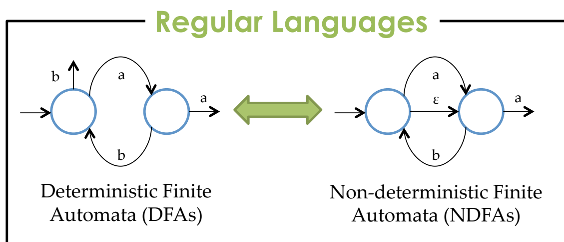
Peter-Michael Osera
http://www.cis.upenn.edu/~posera
posera@cis.upenn.edu

Regular Expressions and Grammars

The World of Computability



The World of Computability



????????

(a U b)*
Regular Expressions

S ➔ aT
Regular Grammars

Regular Expressions

A regular expression (regex) is a string over Σ:

- <u>ε</u>, <u>ø</u>, and a ∈ Σ are regular expressions
- αβ is a regex (α, β are regexes)
- α U β is a regex (α, β are regexes)
- α* is a regex (α is a regex)

Regexes *match* patterns of strings.

Real World Regexes

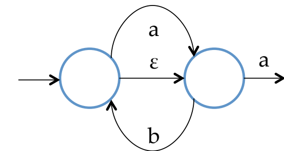
Many different implementations:

- Unix: http://www.grymoire.com/Unix/Regular.html
- Python: https://docs.python.org/2/library/re.html
- Javascript: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Differ in syntax, *syntactic sugar*, etc. but distillable to our core!

For fun, when regexes go wrong: http://ex-parrot.com/~pdw/Mail-RFC822-Address.html

$(a \cup b)^*$??



Regular Expressions

Non-deterministic Finite Automata (NDFAs)

Interpreting Regular Languages

The language of a regex α, denoted $L(\alpha)$ is:

- $L(\underline{\emptyset}) = \emptyset$
- $L(\underline{\varepsilon}) = \{ \varepsilon \}$
- $L(a) = \{ a \}$
- $L(\alpha\beta) = L(\alpha)L(\beta)$ (i.e., *language concatenation*)
- $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$
- $L(\alpha^*) = L(\alpha)^*$ (i.e., the *Kleene star operator)*

Regular Expressions

A regular expression (regex) is a string over Σ:

- $\underline{\varepsilon}$, $\underline{\emptyset}$, and $a \in \Sigma$ are regular expressions
- $\alpha\beta$ is a regex (α, β are regexes)
- $\alpha \cup \beta$ is a regex (α, β are regexes)
- α^* is a regex (α is a regex)

Regexes *match* patterns of strings.

Regex ➔ NDFA

Given a regex α, produce a NDFA D s.t. L(α) = L(D)

- A proof by *structural induction* on α's shape.
- Proceeds by case analysis on the possible forms that α can take.
- When α contains sub-regexes (e.g., $\alpha = \beta \cup \gamma$), assume that we have NDFAs for β and γ.

DFA ➔ Regex

Given a DFA D, produce a regex α s.t. L(D) = L(α)

- Generalize DFA to allow regexes on transitions.
- *Canonicalize* DFA structure (preserving meaning)
- Repeatedly *rip out states* until only the start and accept state remains.
  - The regex on the transition is the final result.

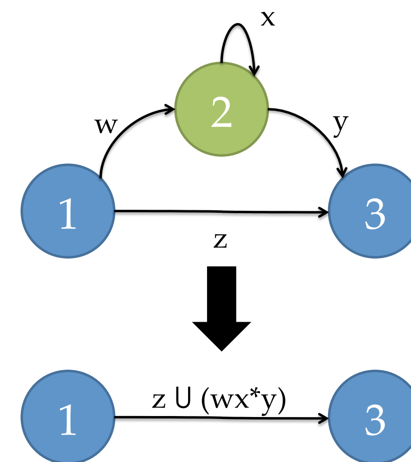
Canonicalization

While preserving the meaning of the DFA:

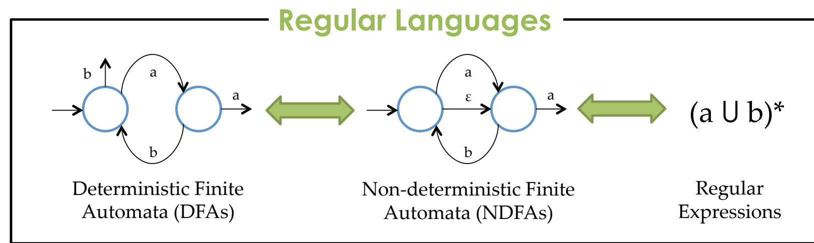
1. Remove unreachable states
2. Make start state have no incoming transitions
3. Make final state have no out-going transitions
4. Give every state exactly one transition to every other state (respecting 2. and 3. above)

Several *gadgets* involved to make this work…

Ripping out States



The World of Computability



????????

S ➔ aT
Regular Grammars

Regular Grammar

A regular grammar is a quadruple (V, Σ, R, S):

- V is the rule alphabet (terminals + non-terminals)
- Σ is the set of terminals (subset of V)
- R is a finite set of rules of the form: X ➔ Y
  - With additional constraints…
- S is the *start symbol* (a non-terminal)

Grammars also match or generate strings.

Regular Grammar Rule Constraints

S ➔ T is a valid regular grammar rule when:

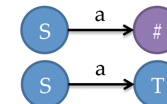
- S is a non-terminal
- T is of the form ε, a, or aU.

S ➔ xT ✔
S ➔ aTb ✗

Regular Grammars ←➔ DFAs

Reg. Grammar ➔ DFA

- Non-terminals become states (S = start state)
- Add one additional accepting state called #
- Rules define transitions and accept states
  - S ➔ ε (S is accepting)
  - S ➔ a
  - S ➔ aT



DFA ➔ Reg. Grammar proceeds similarly…

The World of Computability

??

??

Regular Languages

b a a

Deterministic Finite Automata (DFAs)

a ε a b

Non-deterministic Finite Automata (NDFAs)

??

??

(a U b)*

S ➔ aT

Regular Expressions

Regular Grammars

??

??

??

Context-free Language