CS5142 Scripting Languages
Fall 2013

Regular Expressions

Acknowledgment

These slides are based on slides and lecture notes of Clark Barrett, Robert Grimm, and Ross Tate.

Syntax and Semantics

Syntax refers to the structure of the language, i.e. what sequences of characters are well-formed programs.

- Formal specification of syntax requires a set of rules
- These are often specified using *grammars*

Semantics denotes meaning:

- Given a well-formed program, what does it mean?
- Meaning may depend on context

This lecture only covers syntax. Semantic analysis is covered in the compilers course.

We now look at grammars in more detail.

Grammars

A *grammar* G is a tuple (Σ, N, S, δ), where:

- N is a set of *non-terminal* symbols
- $S \in N$ is a distinguished non-terminal: the *root* or *start* symbol
- Σ is a set of *terminal* symbols, also called the *alphabet*. We require Σ to be disjoint from N (i.e. $\Sigma \cap N = \emptyset$).
- δ is a set of rewrite rules (productions) of the form:

$$\mathrm{ABC}\ldots \rightarrow \mathrm{XYZ}\ldots$$

  where $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}$ are terminals and non-terminals.

Any sequence consisting of terminals and non-terminals is called a *string*.

The *language* defined by a grammar is the set of strings containing *only* terminal symbols that can be generated by applying the rewriting rules starting from S.

Grammars

Consider the following grammar G:

- $N = \{S, X, Y\}$
- $S = S$
- $\Sigma = \{a, b, c\}$
- δ consists of the following rules:
  - $S \to b$
  - $S \to XbY$
  - $X \to a$
  - $X \to aX$
  - $Y \to c$
  - $Y \to Yc$

Some sample derivations:

- $S \to b$
- $S \to XbY \to abY \to abc$
- $S \to XbY \to aXbY \to aaXbY \to aaabY \to aaabc$

The Chomsky hierarchy

- Regular grammars (Type 3)
  - All productions have a single non-terminal on the left and a terminal and optionally a non-terminal on the right
  - Non-terminals on the right side of rules must either always preceed terminals or always follow terminals
  - Recognizable by finite state automaton
- Context-free grammars (Type 2)
  - All productions have a single non-terminal on the left
  - Right side of productions can be any string
  - Recognizable by non-deterministic pushdown automaton
- Context-sensitive grammars (Type 1)
  - Each production is of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$,
  - A is a non-terminal, and α, β, γ are arbitrary strings (α and β may be empty, but not γ)
  - Recognizable by linear bounded automaton
- Unrestricted grammars (Type 0)
  - No restrictions
  - Recognizable by turing machine

Regular expressions

An alternate way of describing a regular language over an alphabet Σ is with *regular expressions*.

We say that a regular expression R denotes the language $[\![R]\!]$ (recall that a language is a set of strings).

Regular expressions over alphabet Σ:

- ϵ denotes $\emptyset$
- a character x, where $x \in \Sigma$, denotes $\{x\}$
- (sequencing) a sequence of two regular expressions RS denotes $\{\alpha\beta \mid \alpha \in [\![R]\!], \beta \in [\![S]\!]\}$
- (alternation) $R|S$ denotes $[\![R]\!] \cup [\![S]\!]$
- (Kleene star) R^* denotes the set of strings which are concatenations of zero or more strings from $[\![R]\!]$
- parentheses are used for grouping
- $R^? \equiv \epsilon|R$
- $R^+ \equiv RR^*$

Regular grammar example

A grammar for floating point numbers:

$$\begin{aligned} \text{Float} &\rightarrow \text{Digits} \mid \text{Digits} \,.\, \text{Digits} \\ \text{Digits} &\rightarrow \text{Digit} \mid \text{Digit}\, \text{Digits} \\ \text{Digit} &\rightarrow 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

A regular expression for floating point numbers:

$$(0|1|2|3|4|5|6|7|8|9)^{+}(.(0|1|2|3|4|5|6|7|8|9)^{+})^{?}$$

The same thing in PERL:

```
[0-9]+(\.[0-9]+)?
```

or

```
\d+(\.\d+)?
```

Deterministic Finite State Automaton

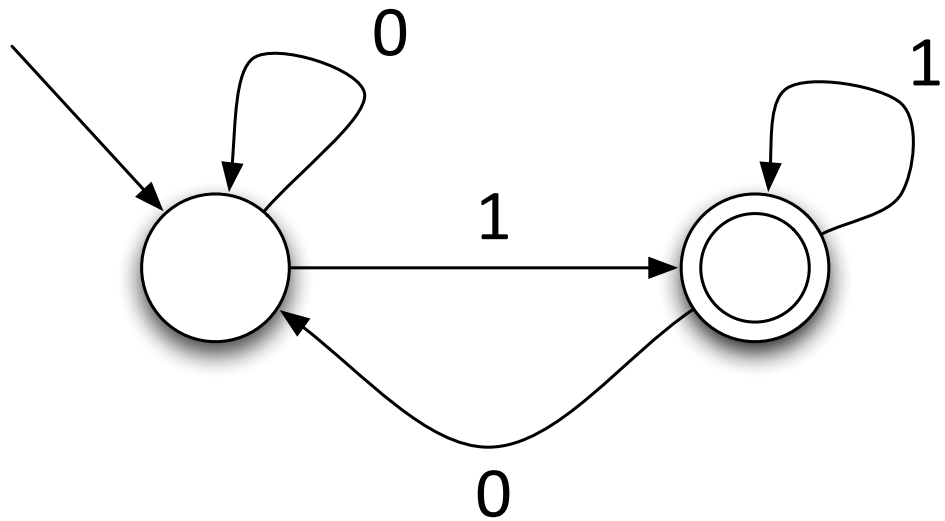
A *deterministic finite automaton* M is a tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finites set of states
- Σ is a finite set of input symbols, also called the *alphabet*.
- δ is a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- q_0 is a initial state $(q_0 \in Q)$
- F is a set of accept states $(F \subseteq Q)$

Deterministic Finite State Automaton

A DFA can be drawn as a labeled graph in which states are nodes, the initial state q_0 is indicated by an incoming edge from outside, other edges are labeled with the corresponding input symbol, and final states in F are marked by nodes with double circles.

For example, consider the following DFA, which accepts only odd numbers expressed in binary, corresponding to the regular expression $(0|1)^*1$:



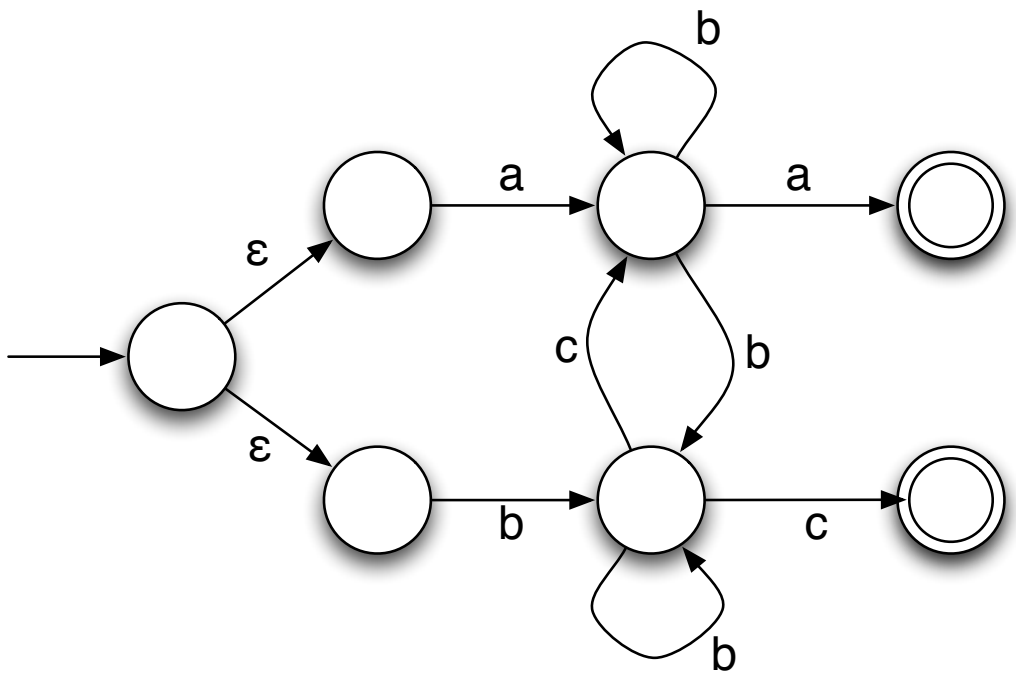
	0	1
q_0	q_0	q_1
q_1	q_0	q_1

The transition function δ can be described as a table. This hints at how you might implement the DFA.

How do you get the δ from a regex?

Non-deterministic Finite State Automaton

An NFA differs from a DFA in that each state can transition to zero or more other states on each input symbol, can also transition to others without reading a symbol. Edges corresponding to not reading a symbol are labeled with ϵ.

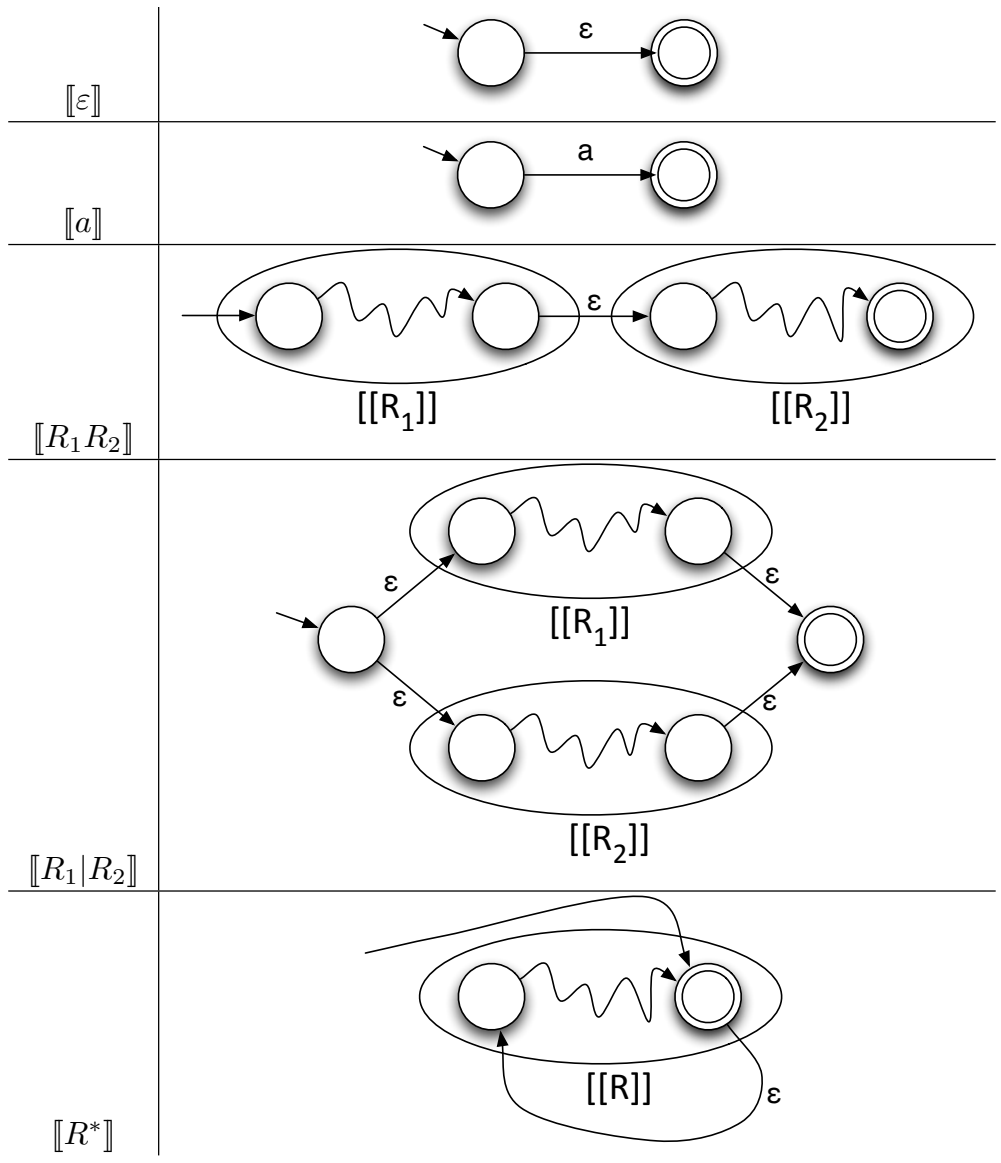


Given an input stream, the NFA accepts if there is *any* way to reach a final state. That is, it has *angelic nondeterminism*. We imagine there is an angel or oracle telling it which transitions to take.

Can you write a regular expression that describes exactly the strings that this NFA accepts?

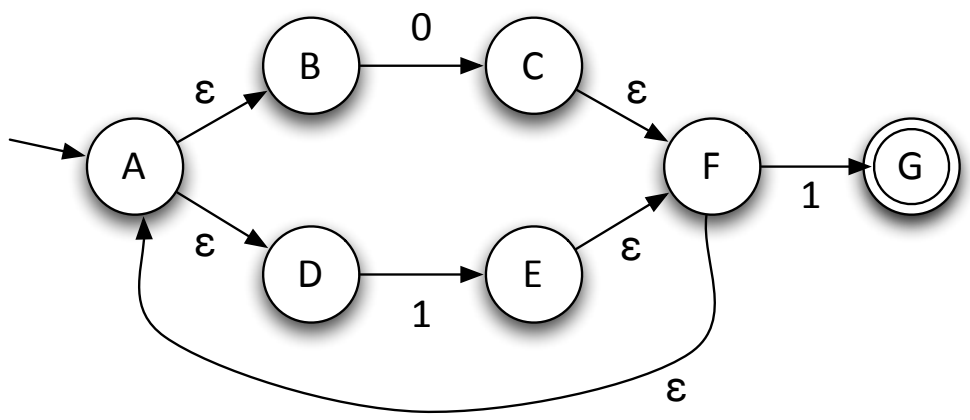
Regex to NFA

Translation works by induction on the structure of a regular expression. That is, we translate an expression by translating its subexpressions. Here, $[\![R]\!]$ means translation:



Regex to NFA Example

The regular expression from before, $(0|1)^*1$, becomes the following NFA:

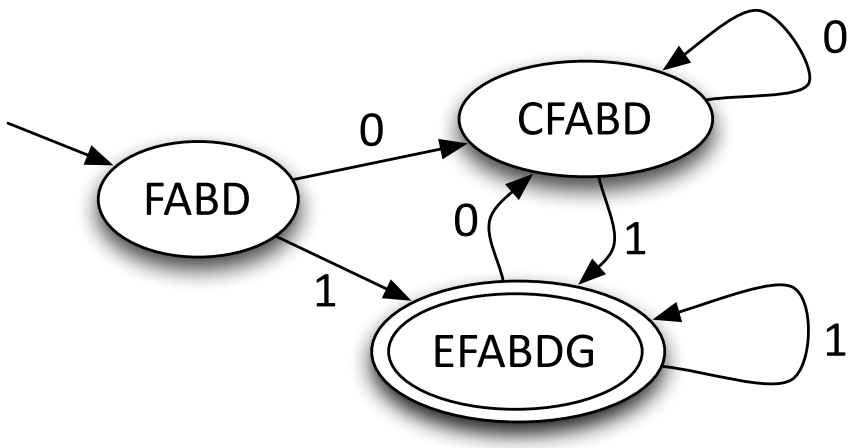


NFA to DFA

- NFAs and DFAs are equivalent (for every NFA, there is a DFA that can express the same language, and vice versa).
- We can convert an arbitrary NFA into a DFA (though the DFA may in general be exponentially larger than the NFA).
- The intuition is that we make a DFA that simulates all possible executions of the NFA.
- At any given point in the input stream, the NFA could be in some set of states. For each set of states the NFA could be in during its execution, we create a state in the DFA.

NFA to DFA

- The $\epsilon - closure$ of a state q is the set of all states reachable from q using zero or more $\epsilon - transitions$.
- The $\epsilon - closure$ of a state $F = \{F, A, B, D\}$
- The initial state of the DFA is the $\epsilon - closure$ of the start state of the NFA
- More-or-less, you keep computing the closure and compressing states



Last Slide

- Next lecture: Return to Perl:
- Context, objects, scripting as glue