

Finite State Automata

Daniel F. Simola

12 May 2005

1 Finite state automata and grammars

1.1 Terminology

Calculus: A logical system used to prove formulae/theorems

- Set of axioms (empty or countably infinite)
- Inference rules for deriving valid inferences

Alphabet Σ : finite set of symbols used as the basis of words and strings

Words W , Strings S : n-tuples of symbols from an alphabet

substring: a contiguous set of symbols

subsequence: a co-linear but not necessarily contiguous set of symbols

prefix: a substring beginning at the first position in a string

suffix: a substring ending at the last position in a string

Language $L(M) = A$: a subset of Σ^* , the set of all n-tuples from Σ (Kleene star), where the subset is specified by a finite state machine M (see below)

Grammar (aka syntax) G : a set of production rules which map symbols from Σ to words/strings in L . ie, a structure that defines a formal language. There are two ways of thinking about grammars (think about similarities with generative vs discriminative classification):

Generative: A set of rules which generates the set of all strings of the language from the alphabet

Analytic: A set of rules which takes a string as input and analyzes (reduces) this input, yielding a yes/no (boolean) output, indicating whether the input string is a member of the language of the grammar. Such a set of rules is called a parser.

A grammar consists of an alphabet (finite set of terminal symbols), a finite set of non-terminal symbols, and a set of production rules mapping a symbol string to another symbol string

1.2 Finite state automata

This leads to the study of finite state machines (FSM) using automata, which are formal models of such machines. A FSM is a five-tuple $(Q, \Sigma, \delta, q_0, F)$ which takes characters from a specific alphabet Σ as input, which are used to transition the machine among one of a finite set of states $Q = \{q_1, \dots, q_m\}$, using a transition function (production function) $\delta : Q \times \Sigma \rightarrow Q$. One of the states is designated as the 'initial' state $q_0 \in Q$, and a subset of states as 'accepting' states, $F \in Q$. Each input changes the state of the machine, and since a FSA has a finite number of states, the transition function is depicted as a transition matrix. A state diagram is a graphical depiction of a particular FSA. (See http://en.wikipedia.org/wiki/Automata_theory for more details.)

A parser is an example of such a machine M , which can be seen to take a string s as input and return whether $s \in A = L(M)$. There exists a hierarchy of languages, where each subsequent language requires

a more powerful grammar (automaton) to parse it. These languages are strictly subsets of each other, in that all context-free grammars can parse all regular expressions, but no regular grammar can parse a single context-free string, etc. (see below).

1.2.1 Determinism vs nondeterminism

Nondeterministic finite state automata (NFA) are simply generalization of deterministic FSA (DFA), where many transitions are possible from each state. Thus every DFA is inherently a NFA. Specifically, where a DFA must have one and only one transition out of each state (outdegree = 1), a NFA may have zero or more outgoing transitions (outdegree > 1). Basically a NFA splits itself into multiple copies and simultaneously visits all possible target states.

In addition a NFA may feature so-called epsilon transitions, where a transition between two states is labeled by ϵ , rather than a symbol in Σ . This means that the FSA splits into multiple copies and follows every one of the ϵ -transitions, in addition to leaving the original machine at the state before the transition point. NFAs can basically do what we would all love to do: if certain copies of the machine wind up not being able to parse a given input string, they just “die”. If at least one copy of the machine remains in an accept state at the end, however, then the machine has parsed the input string. Effectively, after running a FSA (NFA and DFA), you can trace back their progress in the form of a tree, where a DFA forms a linear chain, and an NFA forms a bifurcating tree. A practical example of these trees are syntax trees, which compilers use to parse source code written in a programming language.

Formally a NFA is the same as a DFA, except the transition function maps a (symbol, letter) pair to a *set* of states, rather than a *single* state: $\delta : Q \times \Sigma \rightarrow P(Q)$, where $P(Q)$ is the power set of Q .

Most importantly, you can safely neglect NFAs because a NFA is equivalent to a DFA.

1.2.2 Regular grammars and beyond

A regular grammar is a grammar which composed of the regular operations (NB A and B are languages):

Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$

Star: $A^* = \{x_1x_2\dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

A language is regular if it is closed under these three regular operations ($L(R) = A$). Formally, we can say R is a regular expression if R is

1. a for some a in an alphabet Σ (ie the language $\{a\}$)
2. ϵ (ie the language $\{\epsilon\}$)
3. \emptyset (the empty language)
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*) , where R_1 is a regular expression.

What is more, a regular grammar and a finite state automata are equivalent beasts. They are simply two ways of describing the same language. Take note, however, that they cannot describe **all** languages. There are more powerful languages, that can, such as context-free grammars (CFG) (pushdown automata), context-sensitive grammars (linear-bounded automata), and recursive grammars (Turing machines). Also, note that most programming languages are parsed by context-free grammars.

The difference with a CFG is that it uses production rules of the form:

$$V \rightarrow w,$$

where V is a non-terminal symbol and w is a string consisting of terminals and/or non-terminals. This is context-free because V can be replaced by w in any context. Thus CFG's can be generated recursively

A Quick Note on Grammars

- **Chomsky Hierarchy**
 - Type 0 grammar: all formal grammar, corresponds to languages that can be parsed by a Turing machine
 - Type 1 grammar (context sensitive): Production rules of the form $\alpha A \beta \rightarrow \alpha \phi \beta$, where A is non-terminal, and α, ϕ, β are strings of terminals and non-terminals
 - Type 2 grammar (context independent): Production rules of the form $A \rightarrow \phi$. (Computer languages, RNA structure)
 - Type 3 grammar (regular): Production rules of the form $A \rightarrow \alpha B$, where A and B are single non-terminal symbol and α is a single terminal symbol. (Perl regular expression, Hidden Markov Models).
- **There is an equivalence relationship between “Computing” and grammars.**

(can describe nested string relationships). Contrast this with a context-sensitive grammar, whose production rules are in the form:

$$\alpha V \beta \rightarrow \alpha S \beta,$$

where α, β , and S are strings of terminals and/or nonterminals. This is context-sensitive because α and β define the context in which V can be replaced by a string of terminals/nonterminals.

Just a brief note about Turing machines. They can solve anything an algorithm can solve. In other words a Turing machine is basically formal automaton describing any algorithm. Notably a Turing machine is considered universal because it can simulate another Turing machine in its operation (cf computer OS emulation). This gets into the question of what *can't* be done with a Turing machine, and it turns out there is something called decidability, which has to do with whether a machine will halt on a particular input (Halting problem). Basically given an input and a Turing machine (algorithm), there is no general method of determining whether the machine will halt on the input. Thus the Halting problem is undecidable. Of course the machine can halt, in which case that (machine, string) pair is decidable. See J. Kim's notes for more information about grammars and Turing machines... oh, and computers are Turing machines.

1.2.3 FSA and finite state Markov models

A finite state Markov model is a stochastic model over a set of finite states described by specifying the transition probabilities from state to state. This is equivalent to FSA and regular grammars.