Foundations of Computer Science

Notes from 02-26-2015

Kenny Roffo

April 21, 2015

Contents

1 Formal Languages

1.1 Definitions

1. A symbol is the basic indivisible entity

   Natural Languages: words, not letters

2. An alphabet is a finite, nonempty set of symbols

   Σ is typically used as the name for the alphabet

   Natural languages: $\Sigma = \{$all words in English$\}$ - Lexicon

3. A string (over Σ) is a finite sequence of symbols (over Σ)

   Properties:

   - length
   - empty string denoted λ
   - concatenation
   - λ identity of concatenation

4. Σ^* is the set of all finite strings over Σ

   e.q. $\Sigma = \{0, 1\}$, $\Sigma^* = \{$all strings of 0 and 1$\}$

   Σ is an alphabet
   Σ^* is defined recursively

   λ is an element of Σ^*, and if $w \in \Sigma^*, a \in \Sigma$, then $wa \in \Sigma^*$

5. A language L is any set of strings formed from a given alphabet Σ
   ϕ is a language over any alphabet
   Σ is a language over Σ
   Σ^* is a language over Σ

1.2 Examples

1. $\Sigma = \{b\}$
   $\Sigma^* = \{\lambda, b, bb, bbb, ...\}$

2. $\Sigma = \{a, b\}$
   $\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, ...\}$
   L is a language over Σ that is defined recursively: ex: $L = \{w | w = a^n b^n, n \geq 1\}$
   (a^n means $aaa...a$ n times)
   Thus $L = \{ab, aabb, aaabbb, aaaabbbb, ...\}$, if $w \in L$, then $awb \in L$

3. Let $L_n = a^n b^n$. Then $L = L_0 \cup L_1 \cup L_2 \cup ...$

4. Let Σ_i be the language of i length strings in the alphabet $\Sigma = \{a, b\}$.
   Then $\Sigma_0 = \{\lambda\}$, $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{aa, ab, ba, bb\}$, ...
   It follows that the language $\Sigma^* = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup ... = \bigcup_{i=1}^{\infty} \Sigma_i$

1.3 What is Σ^*

Concatenation - Making new strings from existing strings. We can also concatenate strings with languages and languages with languages. If L_1 and L_2 are languages, then $L_1 L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$

ex: Let $L_1 = \{$in,out$\}$ and $L_2 = \{$law,door,ward$\}$.
Then $L_1 L_2 = \{$inlaw,outlaw,indoor,outdoor,inward,outward$\}$

Σ^* is the set of all strings made from the alphabet Σ. But why Σ^*?
Σ^* is the result of concatenating Σ with itself zero or more times.
Σ^+ is the result of concatenating Σ with itself one or more times.
This is called the positive closure of Σ.

2 Regular Expressions

2.1 What is a Regular Expression?

A regular expression (regex) is a way to specify patterns for strings using union (or), concatenation, and *

A regex over Σ is defined:
Basis: Every $a \in \Sigma$ is a regex over Σ
Recursive: If u and v are regex over Σ then $u|v$, uv, and u^* are all regex over Σ

Here, | means or and * means 0 or more. When in doubt, use parentheses.

grep - general regular expression parser - a Unix command which searches a file for a pattern defined by a regex.

Let $X = \{a, ab, aba\}$ and $Y = \{b, bb\}$. Then

- $XY = \{ab, abb, abb, abbb, abab, ababb\}$ (Concatenation)
- $X|Y = \{a, ab, aba, b, bb\}$ (Like union)
- $X^* = \{a, aa, aaa, ..., aab, ab, abab, ababab, ..., aab, aaba, aaab, ababa, ...\}$
  (All possible strings from 0 or more concatenations)
- $ababa \in X^*$
- $ababa \in XY^*$
- $ab(ab)^*a$ is a regex that matches $ababa$

3 Finite State Machines

3.1 The Vending Machine

Consider a vending machine which contains Jelly beans and Gum. The Machine has inputs

- N - 5 cents
- D - 10 cents
- J - Jelly Bean (Costs 20 cents)
- G - Gum (Costs 15 cents)

These can be represented by $\Sigma = \{N, D, J, G\}$. The machine also has outputs

- b - beep when money is added
- j - jelly bean dispensed
- g - gum dispensed

Design a Finite State machine - a machine with a finite number of "things to remember" This vending machine has to "remember":

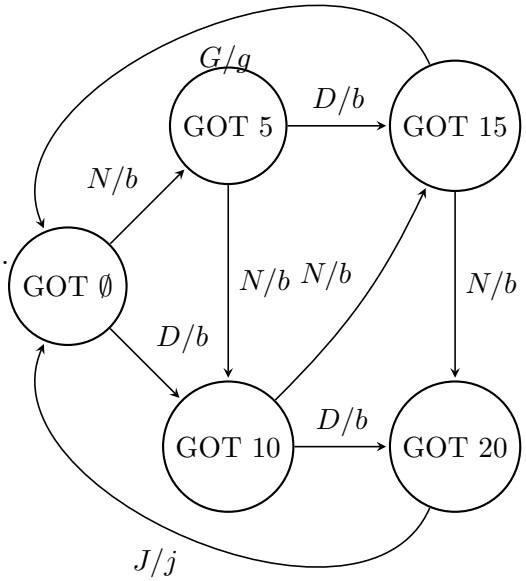
- total money deposited (but not he order in which coins are desposited)
- which product is selected

States are drawn with circles and named on the inside:

For our Vending Machine the states are described by how much money is in the machine, and the transitions represent money input, or a purchase or gum or a jelly bean. The states live in the set $Q = \{GOT\emptyset, GOT5, GOT10, GOT15, GOT20\}$. State transitions are defined by a function $\delta : Q \times \Sigma \to Q$. As an example,

$$(GOT5)(N) \to GOT10$$

Here is an example of a state transition diagram.



3.2 Finite State machine

A finite state machine has one or more states. For the vending machine in the previous section the inital and final states happen to be the same. Any path in the state transition diagram is called a computation. Any path in a finite state machine which starts in the initial state and ends in the final state is called an acceptable path.

A finite state machine must have:

- An input mechanism
- A computing mechanism
- An output mechanism

We use ordered pairs to reference what has happened in the fsm. Referring back to the vending machine, here is an example:

$(GOT5, NNNDG) \to (GOT10, NNDG) \to (GOT15, NDG) \to (GOT20, DG) \to$
$(GOT20, G) \to (GOT\emptyset, \lambda)$

The first part of the pair represents the current state while the second represents the input. Each pair is called an instantaneous configuration.

A finite state machine is defined by a 5-tuple: $M = (Q, \Sigma, \delta, q_0, F)$:

- Q: finite, nonempty set of states
- Σ: alphabet (finite, nonempty set of symbols)
- δ: $Q \times \Sigma \to Q$ is the state transition function
- q_0: initial state
- $F \subset Q$: set of final states

3.3 The Turing Machine

Alan Turing was one of the most influential scientists of the 20^{th} century. One of his contributions was the idea of a Turing Machine.

A Turing Machine is a machine that can solve any computable problem. In other words, if a problem is able to be computed, a Turing Machine can solve that problem.

The Wikipedia definition of a Turing machine:
A **Turing machine** is a hypothetical device that manipulates symbols on a strip of tape according to a table of rules. Despite its simplicity, a Turing machine can be adapted to simulate the logic of any computer algorithm, and is particularly useful in explaining the functions of a CPU inside a computer.

The Man, the Wolf, the Goat and the Cabbage

Consider this classic problem:

A man once had to travel with a wolf, a goat and a cabbage. He had to take good care of them, since the wolf would like to taste a.piece of goat if he would get the chance, while the goat appeared to long for a tasty cabbage. After some traveling, he suddenly stood before a river. This river could only be crossed using the small boat laying nearby at a shore. The boat was only good enough to take himself and one of his loads across the river. The other two subjects/objects he had to leave on their own. How must the man row across the river back and forth, to take himself as well as his luggage safe to the other side of the river, without having one eating another?

A table can be constructed similar to show the possible different states. States that are not allowed are crossed out. The route I chose to take is in bold, and the next possible states from there are listed on the next line. Note that there are infinitely many solutions. I chose my solution as it seemed the most logical to me on the fly.

	M	W	G	C
MWGC-	WGC-M	~~GC-MW~~	**WC-MG**	~~WG-MC~~
WC-MG	**MWC-G**	X	MWGC-	X
MWC-G	WC-MG	C-MWG	X	**W-MGC**
W-MGC	~~MW-GC~~	X	**MWG-C**	MWC-G
MWG-C	~~WG-MC~~	**G-MWC**	W-MGC	X
G-MWC	**MG-WC**	MWG-C	X	MGC-W
MG-WC	G-MWC	X	**-MWGC**	X

3.4 Non-deterministic Finite State Machines

Let $\Sigma = \{a, b\}$. Design a finite state machine that accepts all strings over Σ that contain the substring bab:

Regular Expression: $(a|b)^*bab(a|b)^*$

δ	a	b
q_0	q_0	q_0, q_1
q_1	q_2	...
q_2	...	q_3
q_3	q_3	q_3

As seen in the table, if the state is q_0 and the input is a, then the machine has two options. We call this non-deterministic, and thus we call the machine a non-deterministic finite state machine. Let M be a non-deterministic finite state machine: Then

$$M = (Q, \Sigma, q, \delta, F)$$

where

$$\delta : Q \times \Sigma \to \rho(Q)$$

Recall that $\rho(S)$ is the power set of S. An alternate way of making the table above would be:

δ	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$

We define a new function, $\delta' : \rho(q) \times \Sigma \to \rho(Q)$

δ_1	a	b
$\{q_0\}$	$\{q_0\}$	$\{q_0, q_1\}$
$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_0, q_1\}$
$\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1, q_3\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\}$
$\{q_0, q_2, q_3\}$	$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$

The sets on the left are determined as the table is constructed based on what appears

on the right. Now we let

$$\begin{aligned}
\{q_0\} &= 0 \\
\{q_0, q_1\} &= 01 \\
\{q_0, q_2\} &= 02 \\
\{q_0, q_1, q_3\} &= 013 \\
\{q_0, q_2, q_3\} &= 023 \\
\{q_0, q_3\} &= 03
\end{aligned}$$

Then our machine becomes $M'(Q', \Sigma, \delta', 0, F)$ with

$$\begin{aligned}
Q' &= \{0, 01, 02, 013, 023, 03\} \\
\Sigma &= \{a, b\} \\
\delta' &: \rho(Q) \times \Sigma \to \rho(Q) \\
F &= \{013, 023, 03\}
\end{aligned}$$

δ'	a	b
0	0	01
01	02	01
02	0	013
013	023	013
023	03	013
03	03	013

This may feel a little uneasy at first, but we now have a deterministic finite state machine (DFA).

3.5 Minimizing a DFA

We have a method for ensuring our DFA is minimal. Let's show this by an example. Consider the DFA from the end of the previous subsection:

δ'	a	b
0	0	01
01	02	01
02	0	013
013	023	013
023	03	013
03	03	013

This DFA is not minimal, so we start by making a partition of the states. One set will be the final states, and the other set will be the rest of the states:

$$\Pi_0 : \ {}^{A}[0\ 01\ 02]\ {}^{B}[013\ 023\ 03]$$

From here we look at which sets each element maps to. For example, when the inputs are a and b, the outputs are elements of the sets A and A for 0, A and A for 01, A and B for 02, etc. Since 0 and 01 map to the same sets for both inputs, they will stay together. However, 02 will branch off into its own set, called a singleton. Note that if elements of A have the same mappings as elements of B they do not become members of the same set at the next level of the process since they came from different sets. This means the next step gives:

$$\Pi_1 : \ {}^{C}[0\ 01]\ {}^{D}[02]\ {}^{E}[013\ 023\ 03]$$

We now execute the process again to have:

$$\Pi_2 : \ {}^{F}[0]\ {}^{G}[01]\ {}^{H}[02]\ {}^{I}[013\ 023\ 03]$$

Finally, we will do this one last time to see that $\Pi_3 = \Pi_2$, and thus the process no longer reduces the partition:

$$\Pi_3 : \ {}^{F}[0]\ {}^{G}[01]\ {}^{H}[02]\ {}^{I}[013\ 023\ 03]$$

Now we define:

$$\begin{aligned} 0 &= F \\ 01 &= G \\ 02 &= H \\ 0123 &= I \end{aligned}$$

And define our minimal finite state machine:

δ	a	b
0	0	01
01	02	01
02	0	0123
0123	0123	0123

3.6 Converting NFA to DFA with ϵ moves

To convert an NFA to a DFA we will compute the ϵ-closure of every state.

ϵ-moves are moves without any input.
$\epsilon - c(q) \equiv \{s | s$ can be reached from q by an ϵ-move$\}$

Consider the NFA with ϵ-moves:

δ	0	1	ϵ
q_0	$\emptyset$	$\emptyset$	$\{e0, e1\}$
$e0$	$\{o0\}$	$\{e0\}$	$\emptyset$
$e1$	$\{e1\}$	$\{o1\}$	$\emptyset$
$o0$	$\{e0\}$	$\{o0\}$	$\emptyset$
$o1$	$\{o1\}$	$\{e1\}$	$\emptyset$

We will find the ϵ-closure of every state:

$$\epsilon - c(q_0) = \{q_0, e0, e1\}$$
$$\epsilon - c(e0) = \{e0\}$$
$$\epsilon - c(e1) = \{e1\}$$
$$\epsilon - c(o0) = \{o0\}$$
$$\epsilon - c(o1) = \{o1\}$$

Now we construct δ' with NFA$\rightarrow$ DFA except add ϵ-closures of all resulting states.

δ'	0	1
$\{q_0, e0, e1\}$	$\{o0, e1\}$	$\{e0, o1\}$
$\{o0, e1\}$	$\{e0, e1\}$	$\{o0, o1\}$
$\{e0, o1\}$	$\{o0, o1\}$	$\{e0, e1\}$
$\{e0, e1\}$	$\{o0, e1\}$	$\{e0, o1\}$
$\{o0, o1\}$	$\{e0, 01\}$	$\{o0, e1\}$

Now we define:

$$\{q_0, e_1, e_0\} = i$$
$$\{o0, e1\} = o0e1$$
$$\{e0, o1\} = e0o1 \text{ and so on...}$$

The DFA is now

δ'	0	1
i	$o0e1$	$e0o1$
$o0e1$	$e0e1$	$o0o1$
$e0o1$	$o0o1$	$e0e1$
$e0e1$	$o0e1$	$e0o1$
$o0o1$	$e001$	$o0e1$

4 Sequential Machines

Sequential machines have outputs, like our vending machine.
State transition: $\delta(p, a) = q \quad \delta : Q \times \Sigma \to Q$
Σ is an input alphabet
$\epsilon(p, a) = o$, p is the state, a is the input symbol, o is the output symbol

Sequential Machine:

- Q: Set of states
- Σ: input alphabet
- Δ: output alphabet
- $\delta : \ Q \times \Sigma \to Q$: state transition function
- $\epsilon : Q \times \Sigma \to \Delta$: output function
- q_0: initial state

Ex:

$M = (Q, \Sigma, \Delta, \delta, \epsilon, q_0)$
$Q = \{A, B, C, D, E, F\}$
$\Sigma = \{0, 1\}$
$\delta : \ Q \times \Sigma \to Q$
$q_0 = A$
$\Delta = \{0, 1\}$
$\epsilon : \ Q \times \Sigma \to \Delta$

δ	0	1
A	E	D
B	F	D
C	E	B
D	F	B
E	C	F
F	B	C

ϵ	0	1
A	0	1
B	0	0
C	0	1
D	0	0
E	0	1
F	0	0

We can combine these two functions:

δ, ϵ	0	1
A	$E, 0$	$D, 1$
B	$F, 0$	$D, 0$
C	$E, 0$	$B, 1$
D	$F, 0$	$B, 0$
E	$C, 0$	$F, 1$
F	$B, 0$	$C, 0$

Question: How do we determine if a sequential machine is minimal?

Answer: We use a sequence of partitions.

$$\Pi_0 : [A\ B\ C\ D\ E\ F]$$
$$\Pi_1 : [A\ C\ E]\ [B\ D\ F]$$

We look at the input and output sequences. Two states, S_1 and S_2 are distinguishable if there is a finite input sequence that causes a different output sequence, depending on whether we use S_1 or S_2. We make the first partition (from Π_0 to Π_1) by comparing the outputs, which come from Δ. In the above example, A, C and E ouptut 0 and 1 when the input is 0 and 1 respectively, so they are grouped together. However, B, D and F output 0 and 0 when the input is 0 and 1 respectively, so they are grouped together. From here we can partition the groups by the same method as for a DFA.

5 Context Free Grammars

5.1 An Example to Start

Let's start with simple english. Consider the sentence:

The girl sees a dog.

We can parse this sentence into its constituent phrases:
"The girl" can is a noun phrase, and "sees a dog" is a verb phrase. However, these phrases can be broken up further. "The" is a determiner. "girl" is a common noun. Also, "sees" is a transitive verb, and "a dog" is a noun phrase, with "a" being a determiner and "dog" being a common noun.

We can say a <sentence> consits of a <noun phrase> followed by a <verb phrase>.

$$
\begin{aligned}
S &\rightarrow NP\ VP \\
NP &\rightarrow DET\ CN \\
NP &\rightarrow PN \\
VP &\rightarrow TV\ NP \\
VP &\rightarrow IV \\
VP &\rightarrow IV\ ADV \\
CN &\rightarrow girl|boy|dog|... \\
PN &\rightarrow Jack|Jill|... \\
DET &\rightarrow a|the|every|... \\
TV &\rightarrow sees|likes|... \\
IV &\rightarrow runs|walks|... \\
ADV &\rightarrow quickly|slowly|...
\end{aligned}
$$

These are called rewrite rules, or production rules, or grammar. They are our syntax for the English Language.

- Syntactic Categories of Grammar

$$S, NP, VP, CN, PN, DET, TV, IV, ADV$$

- Lexical Items (Lexicon, Terminal Symbols)

5.2 Grammar Rules

Grammar Rules can be used in two different ways:

1. Parse a given sentence into its constituents.

2. Generate or derive a setence using production rules.

Given a set of grammar rules we can derive some particular string of all terminal symbols:

$$\begin{aligned} S &\implies NP\ VP \\ &\implies PN\ VP \\ &\implies Jack\ VP \\ &\implies Jack\ IV\ ADV \\ &\implies Jack\ walks\ ADV \\ &\implies Jack\ walks\ slowly \end{aligned}$$

5.3 Context

Definition: We have some string uAv where u is a prefix of A, and v is a suffix of A. u and v define the context, or environment of A.

A (now without u and v) is context-free. There is no prefix or suffix of A which would establish a context for A.

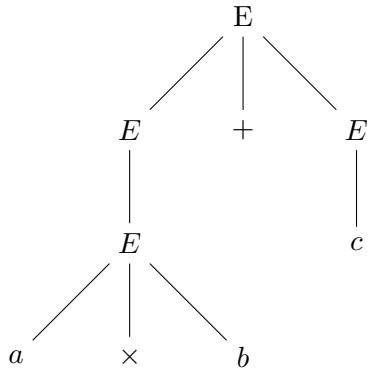
Programming Languages are context-free

Consider a mathematical expression: $(x + y) \times 3$

$$E \to E \times E | E + E | (E) | number | x | y$$

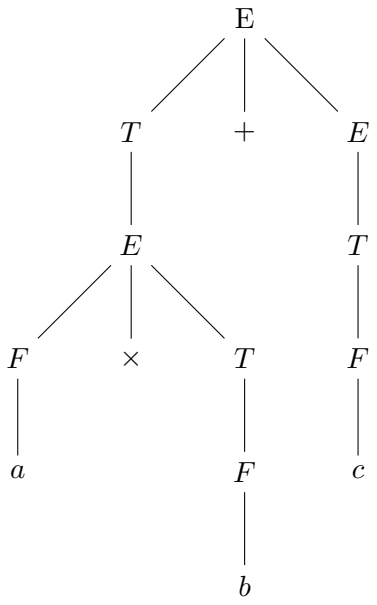
This is the definition of an expression (represented by E).

Consider the expression $a \times b + c$.



$$E \to E + T | E - T | T$$
$$T \to F \times T | T$$
$$F \to (E) | a | b | c$$

We will now define not only expressions, but terms (T) and factors (F). These are a clever way of dealing with the order of operations. It's actually quite beautiful. The "things" which are most specific are "done" first. In other words, since F is the most specific, all factors will always be dealt with first. Then terms, and then expressions. Since expressions become factors (the most specific) by putting them in parentheses, this is how we can see that parentheses truly make their contents be evaluated first.



We can think of the compiler reading an expression as right recursive (but beware, not all compilers are right recursive. It is best to always use parentheses). By right recursion we mean that the reading of the expression will end up being right to left. To try and explain through text, you can look at the definition of T above. Every factor is determined, and the term must be determined before the multiplication is carried out. This means in $a * b * c$ we have the factor a "waiting" for the term $b * c$ to be determined before anything else happens.

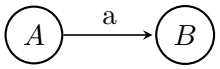
5.4 Regular Grammars

Context free grammars that contain production rules only of the form:

$$A \to aB$$
$$A \to \lambda$$

are called **regular grammars**.

Regular grammars form a subset of the context free grammars, which are exactly the regular languages. We can imagine a correspondence between regular grammars and NFAs:

$A \to aB$ corresponds to 

$A \to \lambda$ corresponds to a final state A: