CS 335 - Week 2 Lecture 1 - January 20, 2011

Crash Intro to Formal Languages (including intro to BNF)

combined version (slides and longer definitions included within the projected notes,
and even some Greek letters and superscripts/subscripts inserted;)

references:
Hopcroft and Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, 1979 (but there is an updated edition with a 3rd author, also)

...as well as the MacLennan course text, Ch. 4

* formal languages - how we describe languages; (meta-languages)
* formally: a programming language is a set of strings (sometimes called sentences) over some finite alphabet of symbols, called terminals
  * the programming language is not necessarily finite, though!
* rules describe how to combine the terminals into well-formed sentences in the programming language - syntax
* programming languages are categorized by the complexity of these syntax rules
* languages that can be defined by REGULAR EXPRESSIONS can be accepted by FINITE AUTOMATA
  RE - regular expressions
  FA - finite automata

RE's are often used to describe TOKENS ("atomic" parts) of programming languages; (also heavily used in Perl, sed, awk, searches, etc.)

BUT -- most programming languages are too complex to describe with RE's;
* We'll find out that many programming languages belong to the language class CONTEXT-FREE LANGUAGES; (CFL's)
  ...described by CONTEXT-FREE GRAMMARS (CFG's)
  (BNF is a form of CFG...)
* (in the interests of time, we are skipping Push-Down Automata - PDA's -- can be used to accept if-statements, looping statements, declarations)

* REGULAR EXPRESSIONS -
* We need some terms first:
what is concatenation on a set of strings?

Let \( \Sigma \) be a finite set of symbols (finite alphabet), and let \( L, L_1, \) and \( L_2 \) be sets of strings from \( \Sigma^* \). The concatenation of \( L_1 \) and \( L_2 \), \( L_1L_2 \), is the set:

\[
\{ xy \mid x \text{ is in } L_1 \text{ and } y \text{ is in } L_2 \}
\]

example: (modified from Sipser, p. 45)

- let \( \Sigma = \{a, b, \ldots z\} \)
- let \( A = \{\text{good, bad}\} \)
- let \( B = \{\text{dog, cat}\} \)
- \( AB = \{\text{gooddog, goodcat, baddog, badcat}\} \)

what is closure on sets of strings?

Let \( L^0 = \{\varepsilon\} \) (the language consisting of the empty string) (DIFFERENT from the empty set!!)

\( L^1 \) is defined as \( L \) concatenated with \( L^0 \) (really, just \( L \), since any string from \( L \) concatenated with the empty string is that string from \( L \)...!)

\( L^2 \) is \( L \) concatenated with \( L^1 \) -- \( L \) concatenated with \( L \), essentially! (all strings made from concatenating 2 strings from \( L \))

\( L^3 \) is \( L \) concatenated with \( L^2 \) -- all strings made from concatenating 3 strings from \( L \)

\[
\ldots
\]

\( L^n \) is the set of all strings made from concatenating \( n \) strings from \( L \)

**Kleene closure** - \( L^* \)

"The Kleene closure (or just closure) of \( L \), denoted \( L^* \), is the set:

\[
L^* = \bigcup_{i=0}^{\infty} L^i
\]

* or, \( L^* \) is the union of \( L^0, L^1, L^2, \ldots L^n \)

**positive closure**: \( L^+ \) - \( L^* \) except \( L^0 \) is not part of the union;

"and the positive closure of \( L \), denoted \( L^+ \), is the set:"

\[
L^+ = \bigcup_{i=1}^{\infty} L^i
\]

* same as \( L^* \), except \( L^0 \) is not included in the unioning of \( L \);

an example for \( A = \{\text{good, bad}\} \)
* example: (modified from Sipser, p. 45)
  * let \( A \), as before, be \{good, bad\}.

  * \( A^* \) contains \{\(\varepsilon\), good, bad, goodgood, goodbad, badgood, badbad, 
goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, \}

  * \( A^+ \) contains \{ good, bad, goodgood, goodbad, badgood, badbad, 
goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, \}

  * (since \( A \) does not contain \(\varepsilon\), then \( A^+ \) does not, either)

* SO... Regular Expressions - longer definitions

* define regular expressions, then; 
  * "Let \( \Sigma \) be an alphabet.

  * The regular expressions over \( \Sigma \) and the sets that they denote are defined recursively as follows:
    1) \( \emptyset \) is a regular expression and denotes the empty set.
    2) \( \varepsilon \) is a regular expression and denotes the set \( \{\varepsilon\} \).
      * [remember: this is the set consisting of the empty string --- this set has one element, the empty string, whereas the empty set has no elements.]
    3) For each \( a \) in \( \Sigma \), \( a \) is a regular expression and denotes the set \( \{a\} \).
    4) If \( r \) and \( s \) are regular expressions denoting the language \( R \) and \( S \), respectively, then:
      (\( r + s \)),
      (\( rs \)), and
      (\( r^* \))
      are regular expressions that denote the sets
      \( R \cup S \),
      \( RS \), and
      \( R^* \),
      respectively."

EXAMPLES:

\( \Sigma = \{0, 1\} \)

11 - represents the language \{11\}

\( (0 + 1)^* \)
represents the closure of the set containing any words from \{0\}
and any words from \{1\} -- the language of ALL strings
of 0's and 1'

\((1 + 10)^*\)
closure of any words from \{1\} and any words from \{10\} --
all words formed by concatenating 1 and 10;
all strings of 0's and 1's that begin with 1
and do not have 2 consecutive 0's;

\(0^*10^*\) - the language \(\{w \mid w \text{ has exactly a single 1}\}\)

* regular expressions often express tokens accepted during
LEXICAL ANALYSIS, often the first "pass" of compiling,
that turns the characters into tokens within the language;

**CONTEXT-FREE GRAMMARS** - describe CONTEXT-FREE LANGUAGES
...can describe features that have a recursive structure;

* what is a CFG?

* finite set of VARIABLES (also called nonterminals
or syntactic categories), EACH of which represents a language;

* the languages represented by the variables are described
recursively in terms of each other, and in terms of
primitive symbols called TERMINALS

* the rules relating variables are called PRODUCTIONS
(sometimes called substitution rules)

* one variable is designated as the START variable --
style rule: this should be the variable on the LHS
of the topmost/first production;

\[ S \rightarrow 0A1 \]
\[ A \rightarrow 1A0 \]
\[ A \rightarrow B \]
\[ B \rightarrow 00 \]

\(S\) - start symbol
the variables here are \(S, A, B\)
the TERMINALS here are \(0, 1\)
these are 4 productions

* you are allowed, if a variable appears on the LHS of more than 1
production, to write them as 1 production with |:

\[ A \rightarrow 1A0 \mid B \]

* derivation: sequence of substitutions to obtain a string
(MUST start from the start symbol!)
(use => to separate "steps" in a derivation)

\[ S \Rightarrow 0A1 \Rightarrow 01A01 \Rightarrow 01B01 \Rightarrow 010001 \]

...this is essentially a proof that 010001 is a string in this language
(CFG's are language GENERATORS...)

* while linguists were studying CFG's, Backus and Naur came up with
BNF to describe Algol-60 --
BNF is CFG notation with minor changes in format, and some shorthand

* so: now let's talk about BNF

* CFG's variables are written in angle brackets in BNF

  <decimal fraction>
  <unsigned integer>

* productions written using ::= instead of ->
  (can be read as "is defined as")

* can use | to write to "combine" productions for the
  same variable;

<integer> ::= +<unsigned integer> | -<unsigned integer>
  | <unsigned integer>

* can use recursion to express sequences;

<unsigned integer> ::= <digit> | <unsigned integer><digit>

* see BNF for an Algol-60 (hardware representation) number

adapted from MacLennan, *Principles of Programming Languages*, 3rd Edition, Chapter 4, Figure 4.1, p. 152

<number> ::= +<unsigned number>
  | -<unsigned number>
  | <unsigned number>

<unsigned number> ::= <decimal number>
  | <exponent part>
  | <decimal number> <exponent part>

<decimal number> ::= <unsigned integer>
  | <decimal fraction>
  | <unsigned integer> <decimal fraction>

<exponent part> ::= E<integer>

<unsigned integer> ::= <digit>
  | <unsigned integer> <digit>

<decimal fraction> ::= .<unsigned integer>

<integer> ::= +<unsigned integer>
  | -<unsigned integer>
  | <unsigned integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

* examples of derivations of a number using this BNF
<number> => <unsigned number>
    => <decimal number>
    => <unsigned integer>
    => <digit>
    => 3

<number> => <unsigned number>
    => <decimal number>
    => <unsigned integer>
    => <digit><digit>
    => 3<digit>
    => 34

* derivation tree - (parse tree)
* write the derivation as a tree, instead;
* the start variable is the root of this tree;
* each substitution (based on a BNF production/rule) adds a level of
  child/children beneath a variable node,
  such that the "children" of that variable's node are what you are
  substituting for that variable;
* when you are done, you'll see that the internal nodes of the resulting tree
  are all variables, and the leaves are all terminals;
* you "read" the string you've just shown is in that language by reading the
  leaves left-to-right;

parse tree for a derivation of 34, showing it is a "legal" number:

```
<number>  
 |         
<unsigned number>  
 |         
<decimal number>  
 |         
<unsigned integer>  
 |         
<unsigned integer> <digit>  
 |         
<digit>  4  
 |         
3
```

34 is an Algol <number>

* sometimes a single parse tree will correspond to several derivations;
  consider the above example: does it really matter whether you substitute
  the first <digit> with 3 first, or the second <digit> with 4 first?

stopping here;