02/03/2010

Recitation #3

Administrative

- Any questions on lecture content?
- Last submission opportunity for Assignment 1a is tonight at 11:59 p.m. (for 20% off)
- Assignment 1b is due February 8th @ 11:59
- Assignment 2 is due February 10th @ 11:59
- Check your Knights email!
- Discussion forum on WebCourses is good interaction and problem solving tool

Assignment 1b

- This builds on assignment 1a, so you should make sure that you get that done

Assignment 2

- Refer to Recitation #2 → assignment was explained there quite explicitly

Worksheet

- An **alphabet** is a finite set of symbols (usually denoted by Σ (sigma)), e.g. $\Sigma = \{0,1,2,3\}$ is the alphabet only containing the numbers 0 – 3; these symbols can be meaningful or abstract; e.g. ASCII, Unicode, English alphabet
- A **string** (= word = sentence) **over an alphabet** is a finite sequence of symbols drawn from the alphabet. All strings of an alphabet are denoted by Σ^*, e.g.

Alphabet	Strings	Σ^*
$\Sigma = \{0,1,2,3\}$	0, 012, 22223, 2131 …	ε , 0, 1, 2, 3, 00, 01, 02, …
$\Sigma = \{a,b,c,\ldots,z\}$	remo, computer, house …	ε , a, b, c, …

- Length of a string s is written as $|s|$; the empty string (ε or λ) has length 0
- A **language** is any countable set of strings over some alphabet, e.g. the language $L = \{\text{a, aardvark, aback, ..., azimuth, azure}\} \in \Sigma^*$ is the set of all word in the dictionary starting with a lower-case "a" and only consisting of lower-case letters, i.e. the alphabet of the language is $\Sigma = \{a,b,c,\ldots,z\}$

- **Regular Expressions** (see recitation #2) describe a set of strings over an alphabet, i.e. regular expressions describe a language, e.g. $(a \mid b)^*$ describes the language consisting of all strings with any number of a's and b's → extended regular expressions used in Flex:
  - Character class $[abc]$ is shorthand for $(a \mid b \mid c)$
  - Ranges $[a-d]$ is shorthand for $(a \mid b \mid c \mid d)$
  - "." Matches any character but newline

- ^ is beginning of line, but $[^s]$ is any one character not in string s, $ is end of line
- $r\{m,n\}$ between m and n occurrences of r

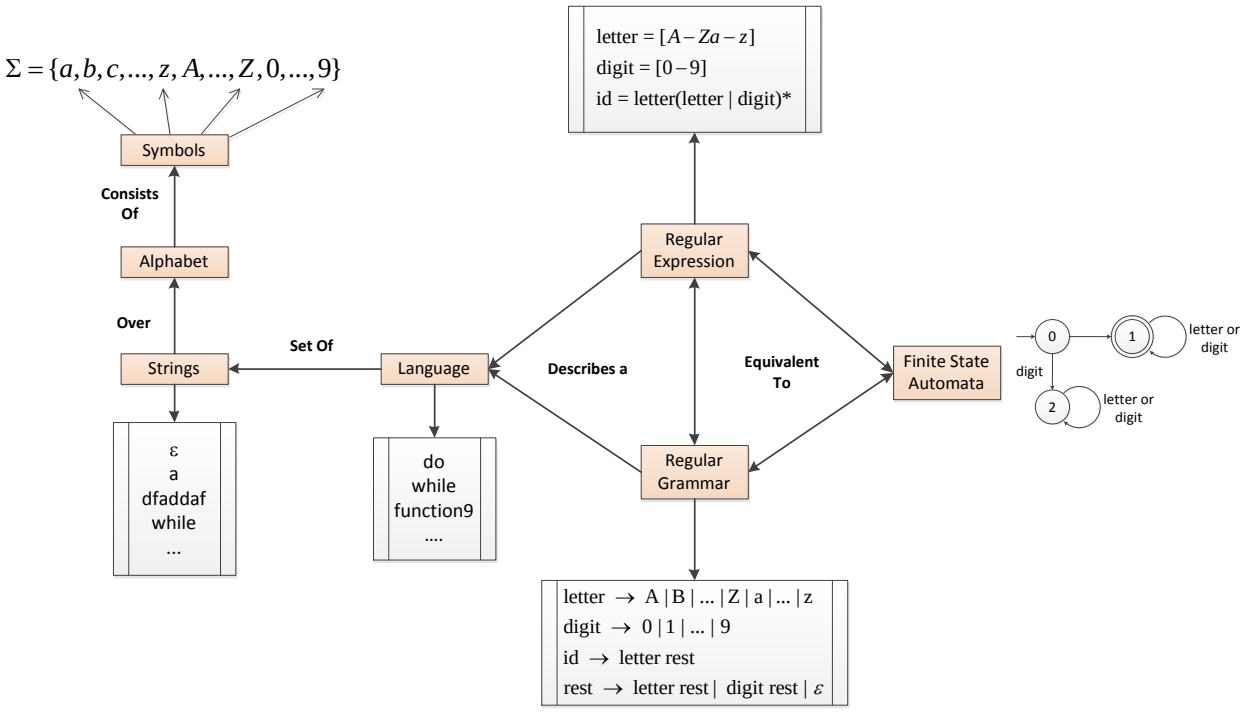
- **Deterministic Finite Automata** (DFA)
  - Formalism for recognizing strings of a language → say "yes" or "no"
  - Accepts input string x if and only if there is some path in the graph from the start state to one of the accepting states, such that the symbols along the path spell out x
  - **A language L is regular if it is the language accepted by some DFA (or can be expressed by some regular expression)**
  - Consists of
    - A finite set of states (usually $Q = \{1,2,3\}$)
    - An input alphabet ($\Sigma = \{[A-Za-z0-9]\}$)
    - A transition function (δ)
    - A start state ($q_0 \in Q = \{1\}$)
    - A set of final states ($F \subseteq Q = \{3\}$)
  - Graphical representation of DFA's
    - Nodes = states
    - Arcs represent transition function
    - Arrow to the start state
    - For each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.
    - Final states indicated by double circle

- **Regular Grammar**
  - Once we know how to express patterns using regular expressions, we would like to find a way to generate all possible strings
- Formal definition: : $G = (V, \Sigma, P, S)$
  - V is a finite set, each element is called a non-terminal character (variable)
  - Σ is a set of terminals, disjoint from V, the alphabet
  - P are production rules with a single non-terminal on the left side and a single terminal or single terminal followed by a single nonterminal on the right side (right-linear grammar)
  - S is the start variable used to represent the whole program. $S \in V$.
  - All production rules of the form (right-linear grammar)
    B → a , where B is a non-terminal and a is terminal
    B → aC
    B → ε
  - For every left-linear grammar there exists an equivalent right-linear grammar
  - Don't mix left and right regular rules, resulting grammar might not be regular!
    S → aA

A → Sb

S → ε

Generates $a^n b^n$, which is not regular!



Context-Free Grammar

- Some languages are non-regular
  - Intuitively, regular languages "cannot count" to arbitrarily high integers
  - e.g. $L = \{0^n 1^n \mid n \geq 1\}$, i.e. 01, 0011, 00001111, …
  - e.g. $L = \{w \mid w \in \{(,)\}^*\}$ and w is balanced, i.e. balanced parenthesis expressions, e.g. (), ()(). (()), (()()), …
  - → Context-free languages
- Finite automata have only finite amount of memory (in form of states) and cannot distinguish infinitely many strings, e.g. for $L = \{0^n 1^n \mid n \geq 1\}$ a finite automaton must remember how many a's it has read when it starts reading b's. Thus it must be in different states when it has read different number of a's and starts reading the first b. But any finite automaton has only finite number of states. Thus there is no way for a finite automaton to remember how many a's it has read for all possible strings $a^n b^n$
- Context-free grammar is regular grammar + allows recursion
- Formal definition: $G = (V, \Sigma, P, S)$
  - V is a finite set, each element is called a non-terminal character (variable)
  - Σ is a set of terminals, disjoint from V, the alphabet

- P is a relation from V to $(V \cup \Sigma)^*$ such that $\exists w \in (V \cup \Sigma)^* : (V, w) \in P$. These are called production or rewrite rules (finite set).
- S is the start variable used to represent the whole program. $S \in V$.

- For $a^n b^n$:
  S → aSb
  S → ab
- For well-formed parentheses:
  S → SS
  S → (S)
  S → ()

  e.g. Starting with S, and applying the rules, one can construct:

  S → SS → SSS → (S)SS → ((S))SS → ((SS))S(S)
  → (((S))S(S) → ((()()))S(S) → ((()()))()(S)
  → ((()()))()(())

- For **syntax analysis** we need something more powerful → context-free languages
- Dr. Hughes was starting to explain grammars in class

Derivation

A sentence generation is called a derivation.

Grammar for a simple assignment statement:

R1 <assgn> → <id> := <expr>
R2 <id> → a | b | c
R3 <expr> → <id> + <expr>
R4 | <id> * <expr>
R5 | (<expr>)
R6 | <id>

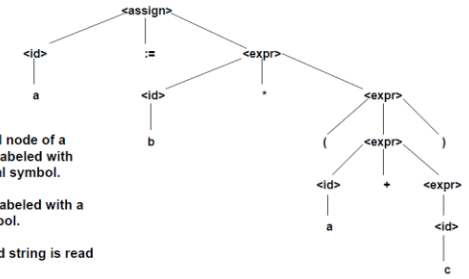
In a left most derivation only the left most non-terminal is replaced

The statement a := b * (a + c)
Is generated by the left most derivation:

<assgn> → <id> := <expr> R1
→ a := <expr> R2
→ a := <id> * <expr> R4
→ a := b * <expr> R2
→ a := b * (<expr>) R5
→ a := b * (<id> + <expr>) R3
→ a := b * (a + <expr>) R2
→ a := b * (a + <id>) R6
→ a := b * (a + c) R2

Parse Trees

A parse tree is a graphical representation of a derivation
For instance the parse tree for the statement a := b * (a + c) is:



Every internal node of a parse tree is labeled with a non-terminal symbol.

Every leaf is labeled with a terminal symbol.

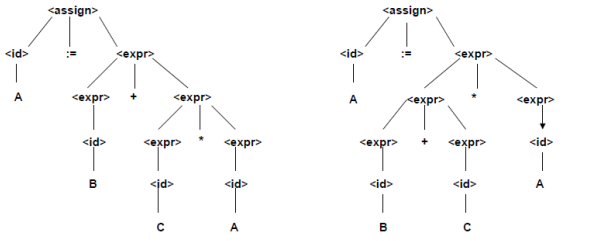
The generated string is read left to right

Ambiguity

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be "ambiguous"

For instance, the following grammar is ambiguous because it generates distinct parse trees for the expression a := b + c * a

<assgn> → <id> := <expr>
<id> → a | b | c
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>



This grammar generates two parse trees for the same expression.

If a language structure has more than one parse tree,
the meaning of the structure cannot be determined uniquely.