

Formal Language Description

I. Syntax

A. Formal description is helpful (definition by example is not sufficient)

1. It facilitates understanding the lang.
2. It can be used to "automatically" create syntax analysis part of compiler; also debuggers

B. Formal Language Theory: Chomsky in 50's

1. A grammar describes a language and consists of:

- a. Start symbol
- b. Set of terminal symbols (alphabet of the language)
- c. Set of non-terminal symbols
- d. Set of rewrite (production) rules

2. E.g. Simple grammar for subset of English [Drake]:

S → NP VP
NP → Art Adj N | Art N | N
VP → V | V NP | V PP
PP → Prep NP
N → book | class | quiz
etc.

3. Can build a parse tree

4. Chomsky Hierarchy

- a. Type 0: recursively enumerable. Left side of rule must be non-empty
- b. Type 1: context-sensitive. $xAy \rightarrow xBy$
- c. Type 2: context-free. Allows only a single non-terminal on l.h.s.
- d. Type 3: regular. r.h.s. may only be a single terminal or single nonterminal followed by a single terminal (left recursive) Note it can't define nested structures.
 - Lexers usually described using a regular lang.,
 - Syntax of prog. lang. usually described using a context-free lang.

C. BNF

1. E.g. A small grammar & a derivation [Seb p.109-110]

Trace derivation by numbering rules and showing where they are applied

- a. Each string in derivation is a *sentential form*
- b. Always replacing left-most non-terminal gives a *left-most derivation*

2. Ambiguous grammar for assignment statement [Seb. pp.110-113]

Grammars describe the hierarchical structure of sentences in the language they describe

- a. Parse Tree - a representation of the hierarchical structure. Every subtree of a parse tree describes one instance of an abstraction in the statement
- b. Syntax analyzers for compilers generate parse trees either explicitly or implicitly (either top-down or bottom-up)
- c. Ambiguity is a problem: Semantics will be determined by syntax (e.g. which operation goes first: + or / ?)
- d. Operator Precedence: transformations *lower* in the tree have precedence (they occur before) over transformations *higher* in the tree (which occur later). An ambiguous parse tree means there is conflicting precedence information. [see overhead]

3. Unambiguous grammar [Seb. p. 114]
 - a. Grammar can be rewritten to consistently have multiplication lower in the tree than addition
 - b. This is done by adding separate abstractions (production rules) for multiplication and addition [See overhead] (essentially adding a separate state)
 - c. See example of ambiguous vs. unambiguous grammar for if-then-else [Sebesta, p. 117]
4. Associativity: Does a grammar maintain proper ordering of, say, addition for $a := a + b + c$;
 (addition is *left associative*)
 - a. The given grammar [see overhead] is *left recursive*, and so further transformations in a derivation for multiple additions always give the left-most additions further down the tree, which is correct.
 - b. Productions for *right associative* operations (e.g. exponentiation) are inherently *right recursive* E.g.

$$\begin{aligned} \langle \text{factor} \rangle &\rightarrow \langle \text{exp} \rangle ** \langle \text{factor} \rangle \\ &\quad | \langle \text{exp} \rangle \\ \langle \text{exp} \rangle &\rightarrow (\langle \text{expr} \rangle) \\ &\quad | \langle \text{id} \rangle \end{aligned}$$

D. EBNF [Seb. p. 119]

1. brackets [] used to indicate an optional part.
2. braces { } used to indicate 0 to infinity repetitions
3. multiple choice options placed in parenthesis () separated by OR |

E. Syntax Graphs