# Formal Language Specification

- Programming languages are only useful if they are "understood" by a computer.
- In order to insure this, programming languages must have:
  - A concise form (syntax), and
  - A concise meaning (semantics)

☞ neither one can be ambiguous.

# Reading

- Chap 2 in MPL

# Formal Language Specification

Language Specifications consist of two parts:

- The <u>syntax</u> of a programming language is the part of the language definition that says what programs look like; their <u>form</u> and <u>structure</u>.
- The <u>semantics</u> of a programming language is the part of the language definition that says what programs do; their <u>behavior</u> and <u>meaning</u>.

# Formal Language Specification

In order to insure conciseness of language specifications we need tools:

- Grammars are used to define the syntax.
- Mathematical constructs (such as functions and sets) are used to define the semantics.

# Grammars

Example: a grammar for simple English sentences.

Start Symbol

```
<Sentence>*       ::= <Noun-Phrase> <Verb> <Noun-Phrase>
<Noun-Phrase>     ::= <Article> <Noun>
<Verb>            ::= loves | hates | eats
<Article>         ::= a | the
<Noun>            ::= dog | cat | rat
```

Production

Non-terminal

Terminal

☞ Grammars capture the structure of a language.

# Grammars

Observations:

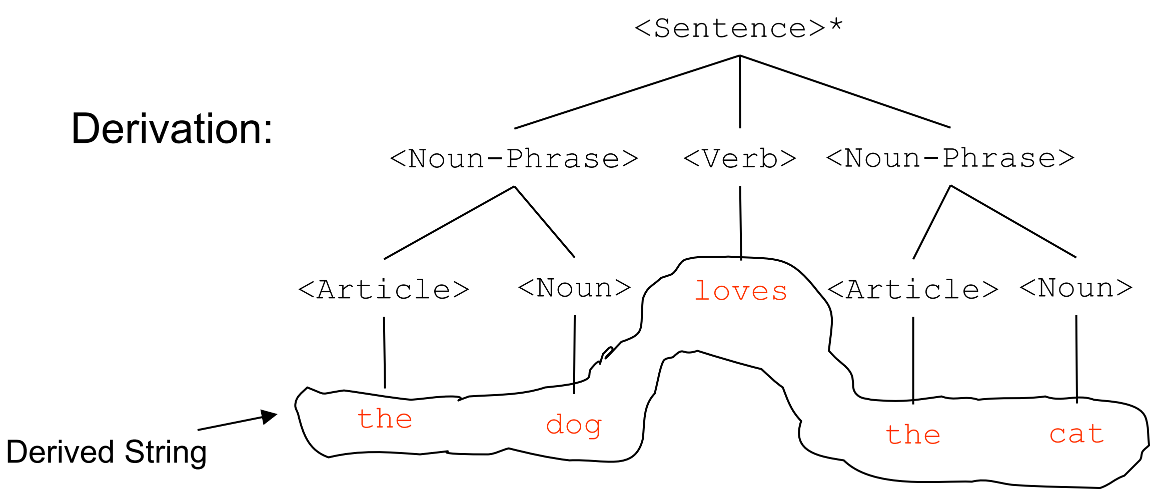
- A grammar consists of a collection of productions.
- Each production defines the "structure" of a non-terminal.
- There are no productions for terminals.
- In a grammar there is a unique non-terminal, the start symbol, that defines the largest structure in our language.

# How do Grammars work?

We can view grammars as rules for building parse trees or derivation trees for sentences in the language defined by the grammar.  In these parse or derivation trees the start symbol will always be at the root of the tree.

```
<Sentence>*          ::= <Noun-Phrase> <Verb> <Noun-Phrase>
<Noun-Phrase>        ::= <Article> <Noun>
<Verb>               ::= loves | hates | eats
<Article>            ::= a | the
<Noun>               ::= dog | cat | rat
```

Derivation:



# How do Grammars work?

Notes:

- A derived string can only contain terminals.
- The language defined by a grammar is the set of all derived strings, formally

  $L(G) = \{\ s \mid s \text{ can be derived from } G\ \}$

  where *G* is a grammar and s is a string of terminal symbols.

# How do Grammars work?

Now we can ask questions as follows:

- Assume we have a grammar *G* and a sentence *s*, does *s* belong to *L*(*G*)?
- In other words, is the sentence *s* a derived string from *G* and, it therefore belongs to *L*(*G*)?

Examples: let *G* be our English grammar,

- Does *s* = “the cat eats a rat” belong to *L*(*G*)?
- Does *s* = “the dog chases the cat” belong to *L*(*G*)?

☞ Show that $s \in L(G)$ by constructing a parse tree.

☞ Show that $s \notin L(G)$ by proving that no parse tree can exist for this string in *G*.

# Take Away

- Programming language specifications consist of two parts: a syntax  and a semantic specification
- We use grammars to specify the syntax unambiguously
- Grammars:
  - Productions
  - Non-terminals
  - Terminals
  - Start symbol
- In order to prove that a string s belongs to L(G) we construct a parse tree
- In order to prove that a string s does not belong to L(G) show that a parse tree cannot exist.