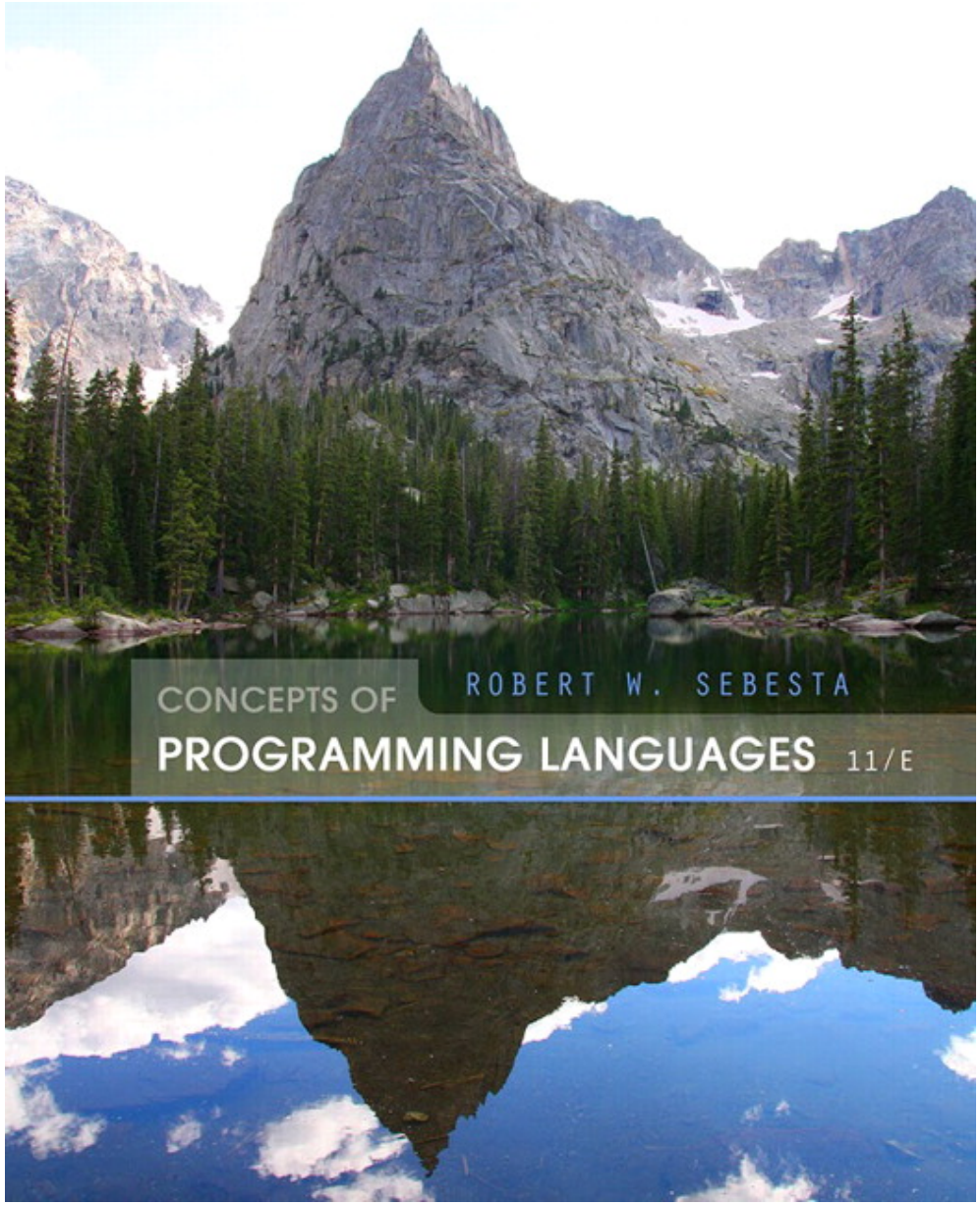
# Chapter 3

## Describing Syntax and Semantics



# Chapter 3 Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

# Introduction

- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

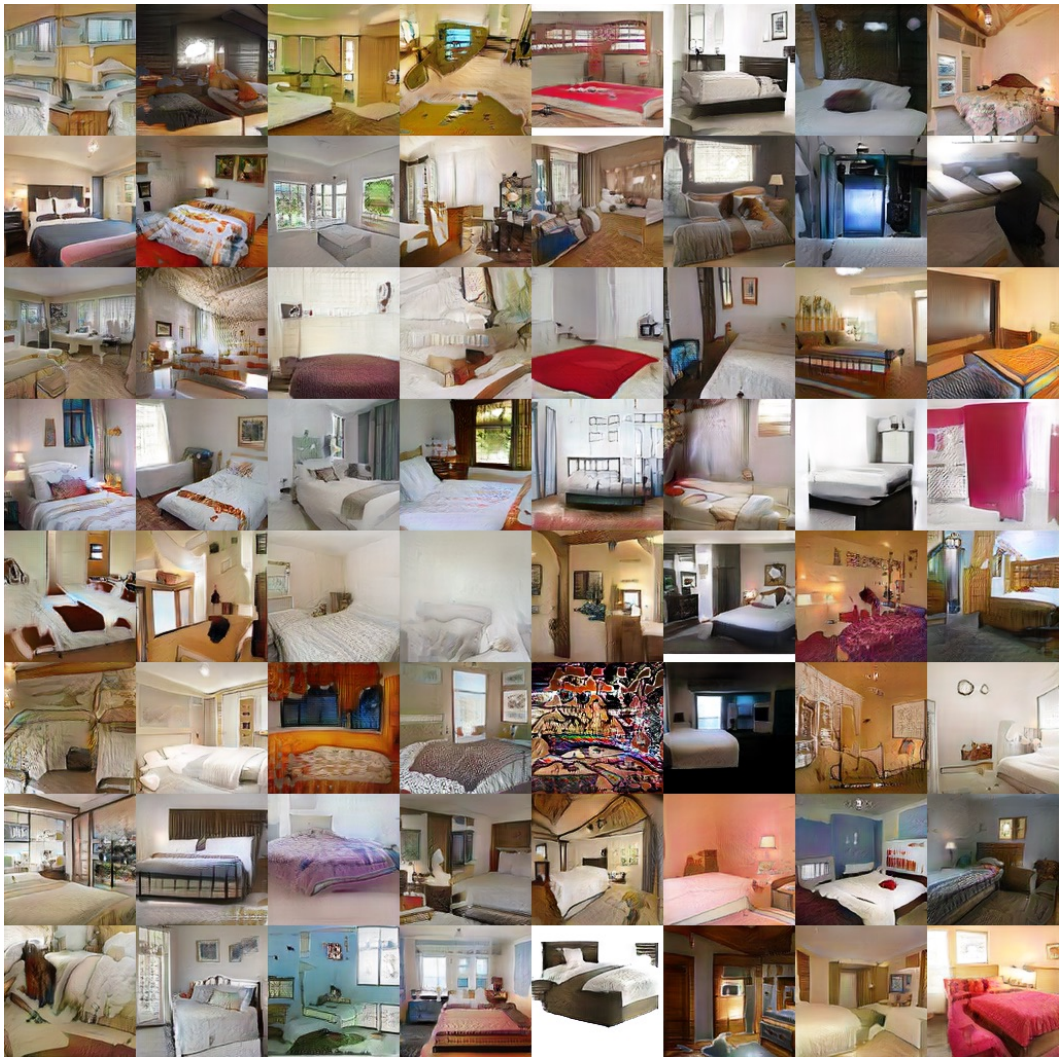
# The General Problem of Describing Syntax: Terminology

- A sentence is a string of characters over some alphabet
- A language is a set of sentences
- A lexeme is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`)
- A token is a category of lexemes (e.g., identifier)

# Formal Definition of Languages

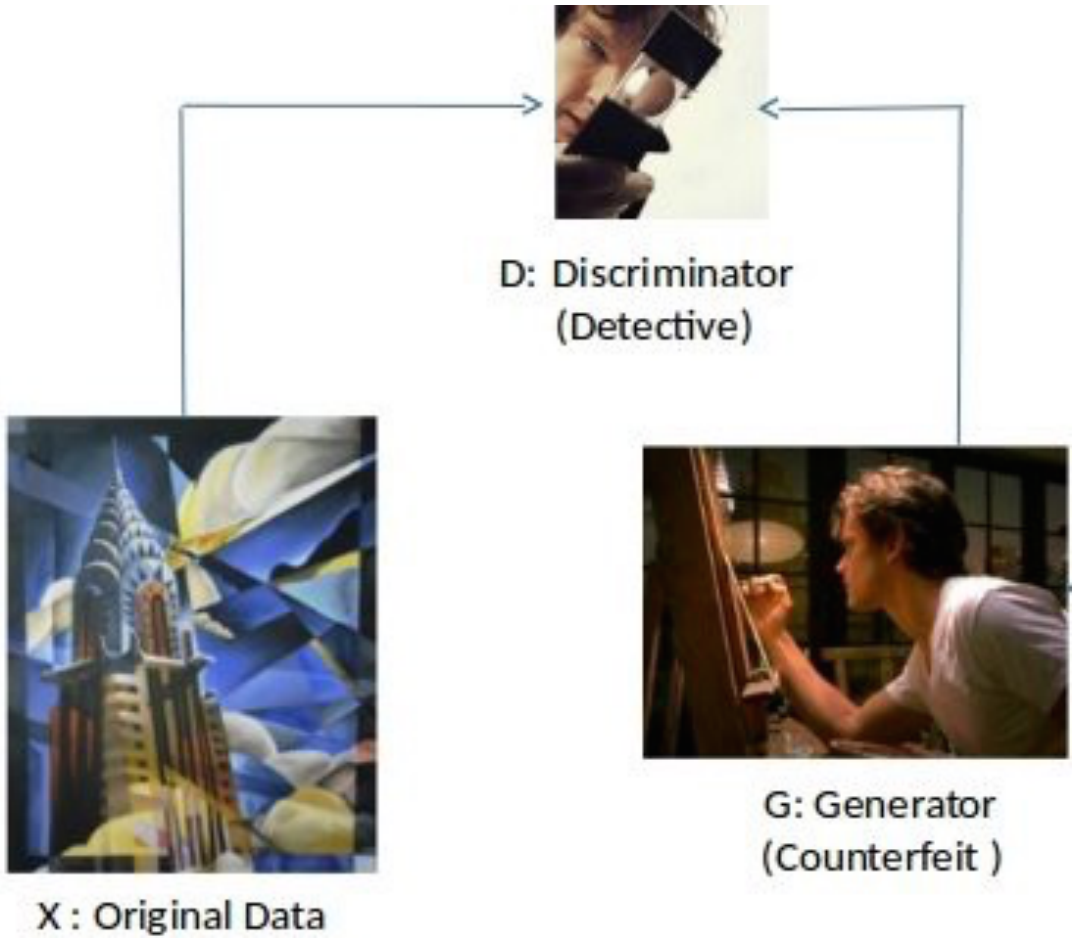
- Recognizers
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
    - Detailed discussion of syntax analysis appears in chapter 4
- Generators
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

# Generating images…



Generative adversarial networks (GAN)
(Goodfellow 2014; Gulrajani et al. 2017)

# Generating images…




Generative adversarial networks

# Natural language (Google translate)

Haruki Murakami compares his own translation from Japanese to English (of Hemingway's Snows of Kilimanjaro") to the 2016 (deep learning) Google translate

NO. 1:
Kilimanjaro is a snow-covered mountain 19,710 feet high, and is said to be the highest mountain in Africa. Its western summit is called the Masai "Ngaje Ngai," the House of God. Close to the western summit there is the dried and frozen carcass of a leopard. No one has explained what the leopard was seeking at that altitude.

NO. 2:
Kilimanjaro is a mountain of 19,710 feet covered with snow and is said to be the highest mountain in Africa. The summit of the west is called "Ngaje Ngai" in Masai, the house of God. Near the top of the west there is a dry and frozen dead body of leopard. No one has ever explained what leopard wanted at that altitude.

from New York Times, Dec 14 2016

# BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s for natural languages
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars

# BNF Fundamentals

- In BNF, abstractions are used to represent syntactic structures (also called nonterminal symbols, or just nonterminals)
- Terminals are lexemes or tokens
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals
- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<assign> → <var> = <expression>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A start symbol is a special element of the nonterminals of a grammar

# BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

# Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> → ident
             | ident, <ident_list>
```

# Example Grammar for small language

```
<program> → begin <stmt_list> end
 <stmt_list> → <stmt>
             | <stmt> ; <stmt_list>
 <stmt> → <var> = <expression>
 <var> → a | b | c
 <expression> → <var> + <var>
              | <var> - <var>
              | <var>
```

# Example Grammar for small language

```
<program> → begin <stmt_list> end
 <stmt_list> → <stmt>
             | <stmt> ; <stmt_list>
 <stmt> → <var> = <expression>
 <var> → a | b | c
 <expression> → <var> + <var>
              | <var> - <var>
              | <var>
```

# Example derivation

<program> => begin <stmt_list> end

- We'll derive A = B + C; B = C with this grammar
- A derivation is a repeated application of rules, starting with the start symbol (in this case program)
- => reads "derives"

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end
=> begin A = B + C ; <var> = <expression> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end
=> begin A = B + C ; <var> = <expression> end
=> begin A = B + C ; B = <expression> end

# Example derivation

<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end
=> begin A = B + C ; <var> = <expression> end
=> begin A = B + C ; B = <expression> end
=> begin A = B + C ; B = <var> end

# Example derivation

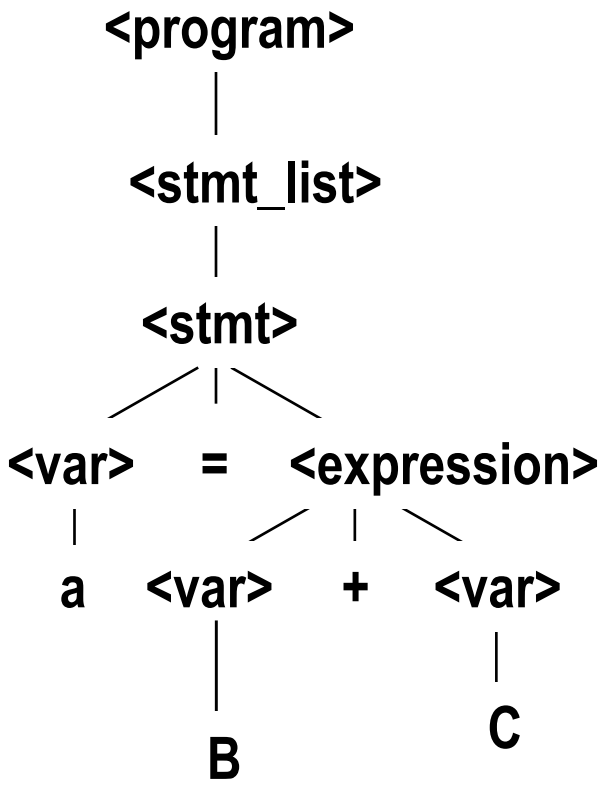
<program> => begin <stmt_list> end
=> begin <stmt> ; <stmt_list> end
=> begin <var> = <expression> ; <stmt_list> end
=> begin A = <expression> ; <stmt_list> end
=> begin A = <var> + <var> ; <stmt_list> end
=> begin A = B + <var> ; <stmt_list> end
=> begin A = B + C ; <stmt_list> end
=> begin A = B + C ; <stmt> end
=> begin A = B + C ; <var> = <expression> end
=> begin A = B + C ; B = <expression> end
=> begin A = B + C ; B = <var> end
=> begin A = B + C ; B = C end

# Derivations

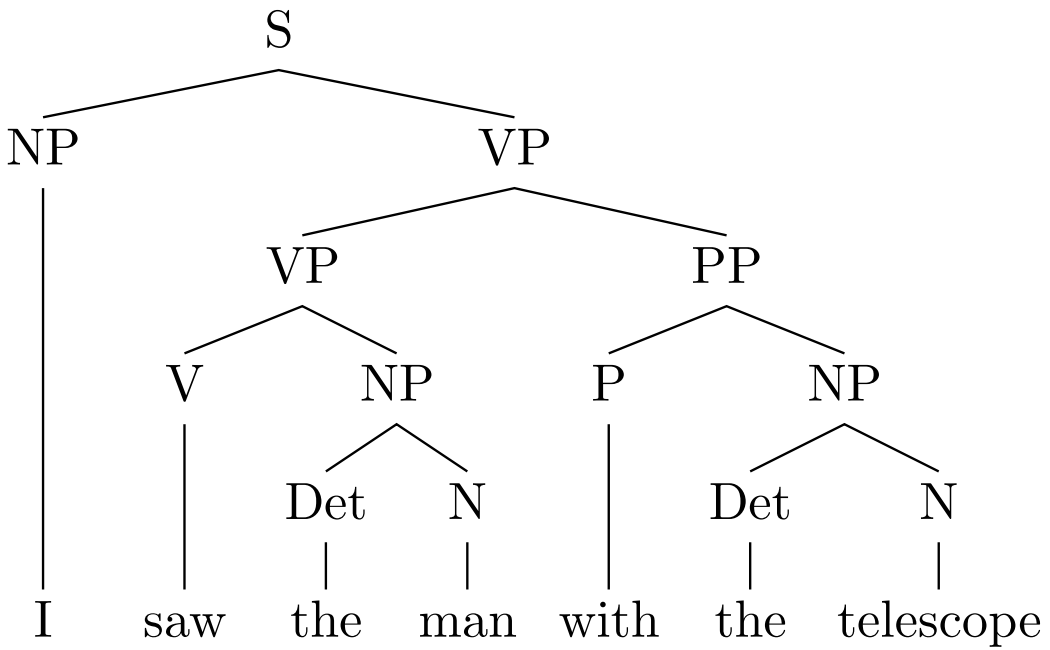
- Every string of symbols in a derivation is called a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

# Parse Tree

- A hierarchical representation of a derivation



# English language example




**PROBABILITY AND STATISTICS IN COMPUTATIONAL LINGUISTICS, A BRIEF REVIEW**

STUART GEMAN* AND MARK JOHNSON*

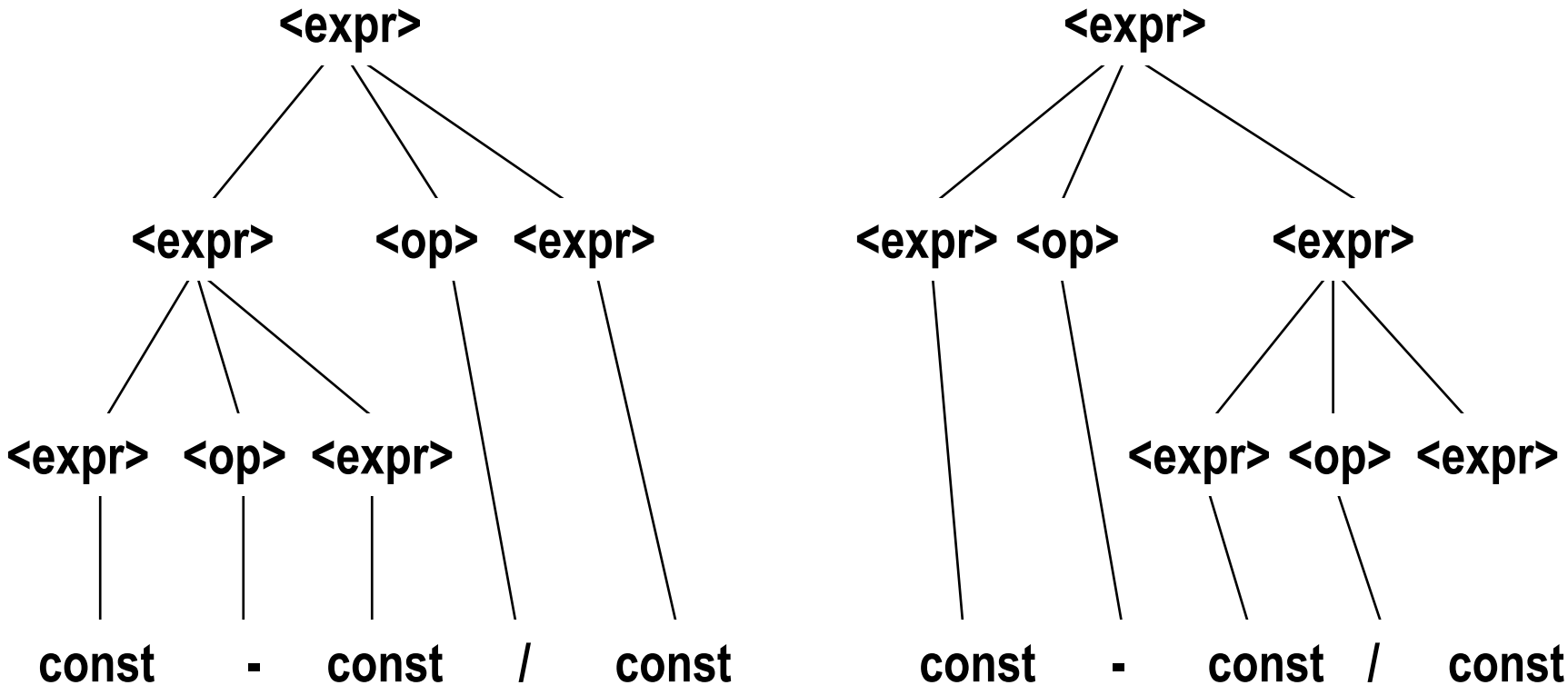
# Ambiguity in Grammars

- A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees
- Problematic for compilers since parse tree, and therefore meaning of the structure, cannot be determined uniquely

# An Ambiguous Expression Grammar

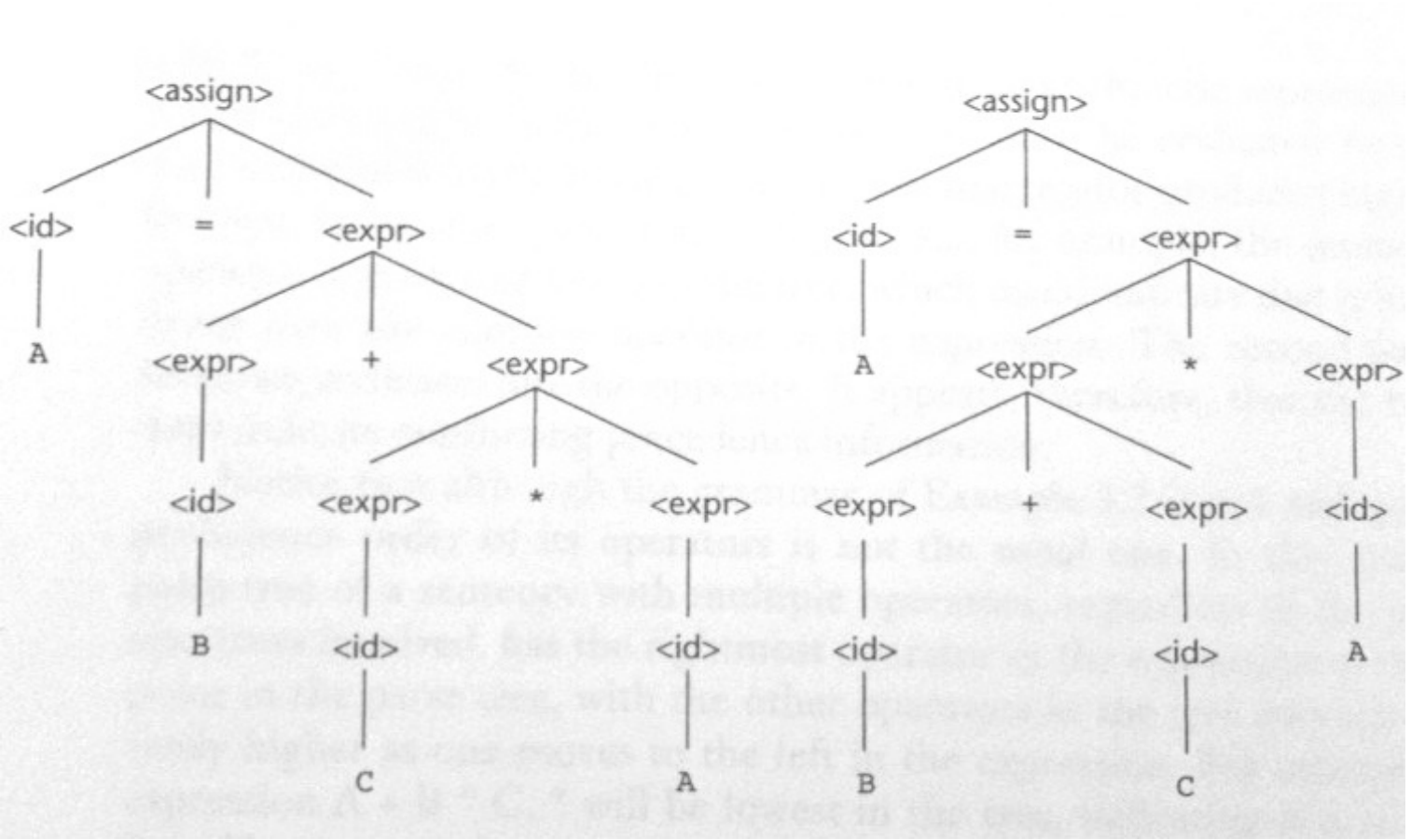
<expr> → <expr> <op> <expr> | const

<op> → / | -

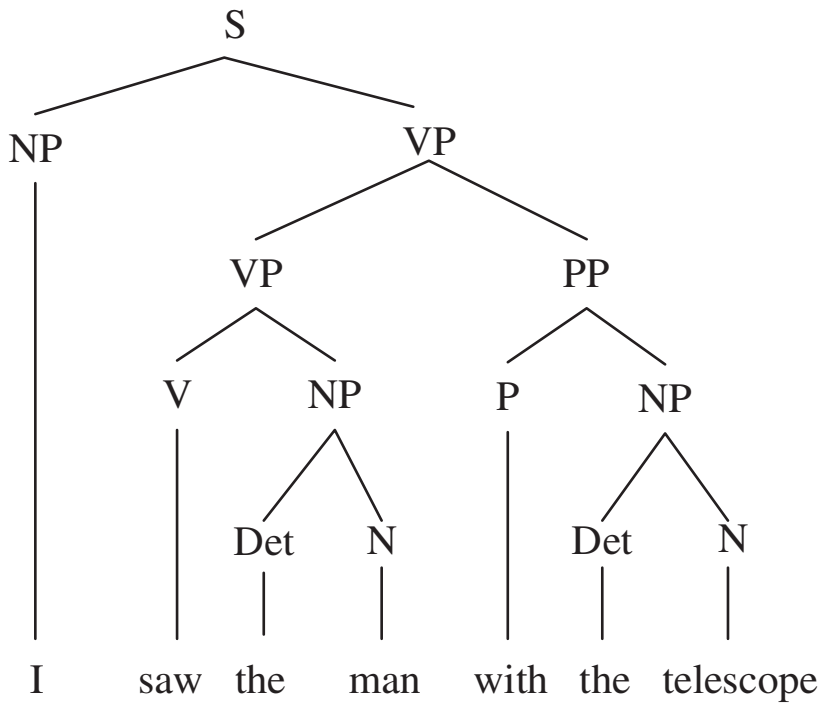


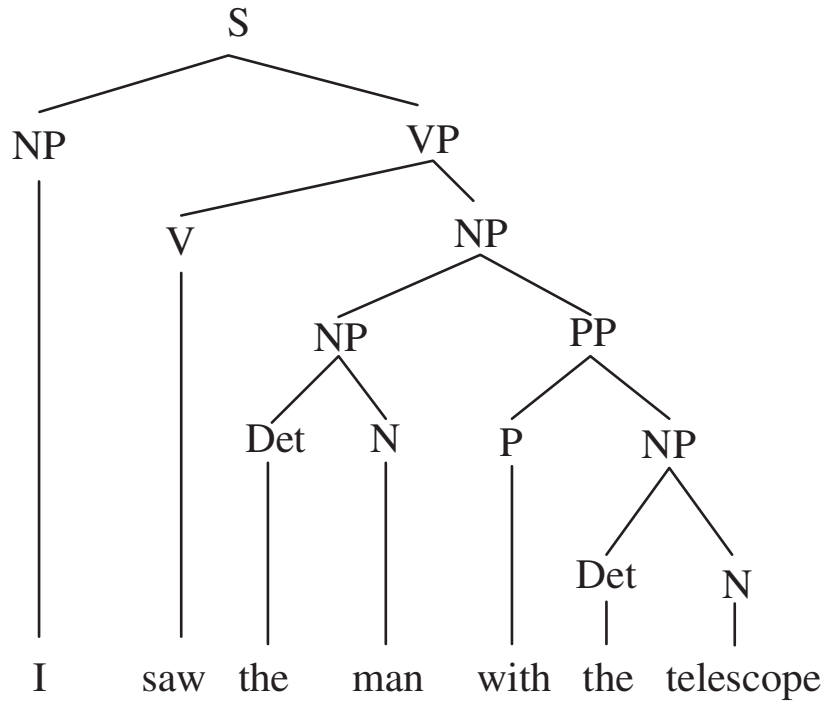
# Example

- 2 parse trees for the sentence A=B+C*A
- Operator precedence
- Conflicting precedence



# English language



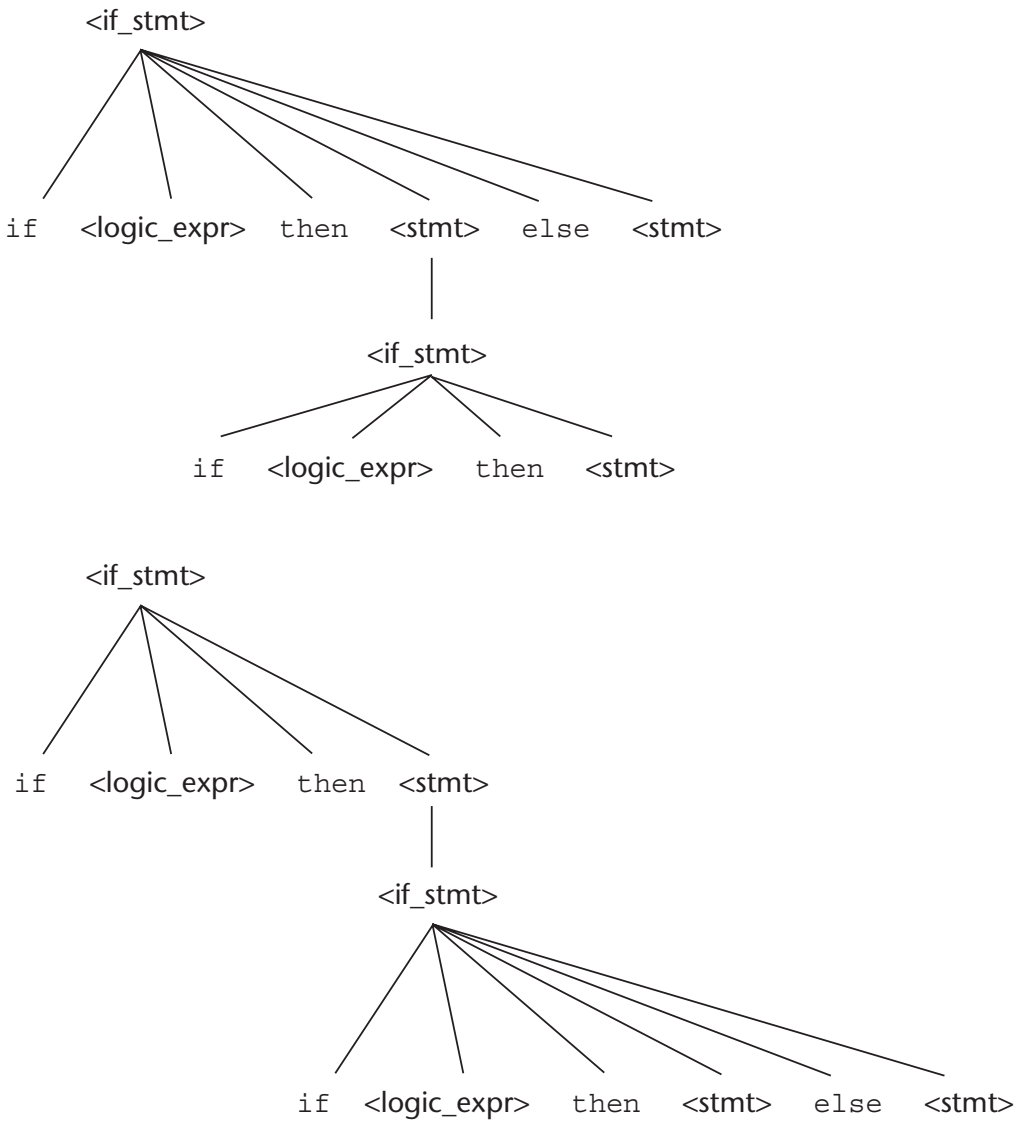

**PROBABILITY AND STATISTICS IN COMPUTATIONAL LINGUISTICS, A BRIEF REVIEW**

STUART GEMAN* AND MARK JOHNSON*

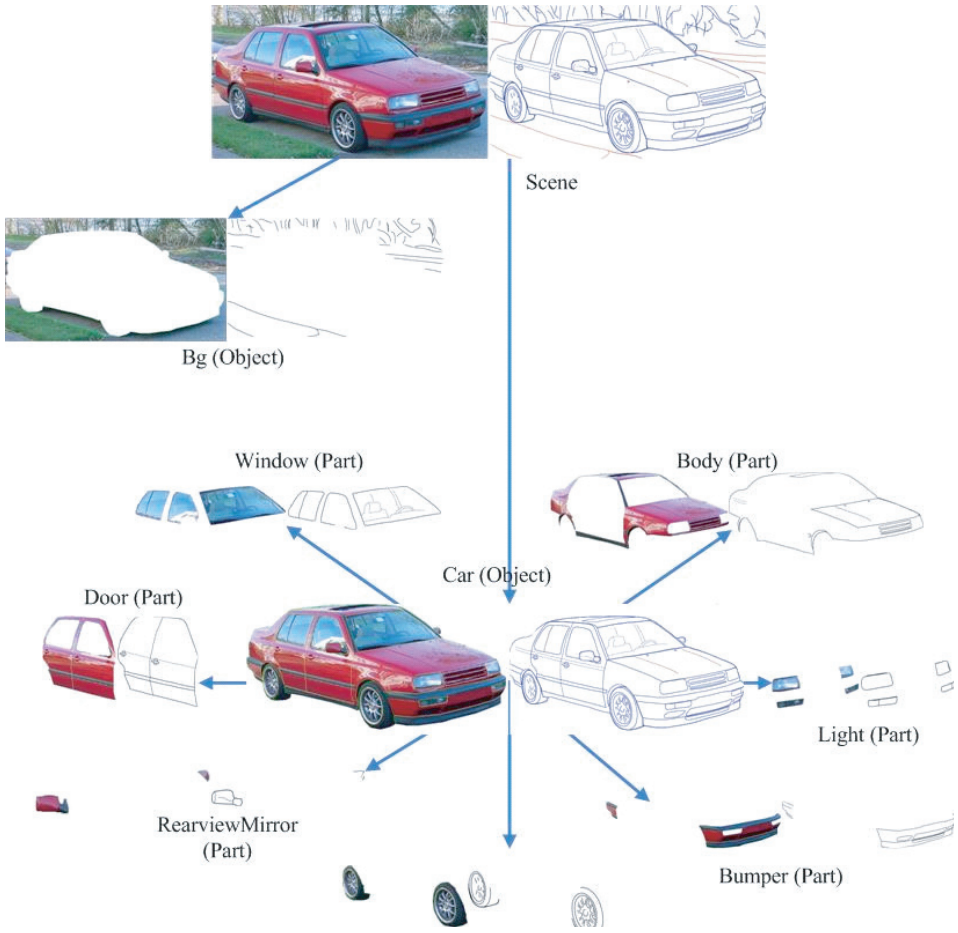
# If else statement

**Figure 3.5**

Two distinct parse trees for the same sentential form



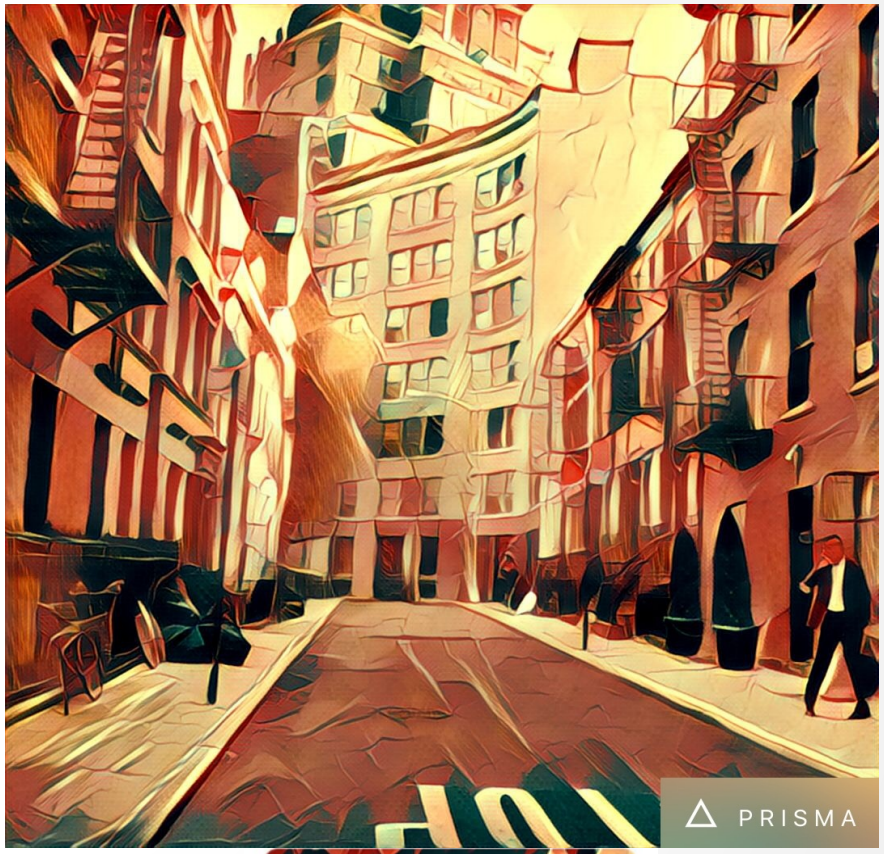
# Image grammars




**A Stochastic Grammar of Images**

Song-Chun Zhu[1,*] and David Mumford[2]

# Generating images…





See also Gatys et al. 2015:
Separating content and style  in a deep network

# Generating images…