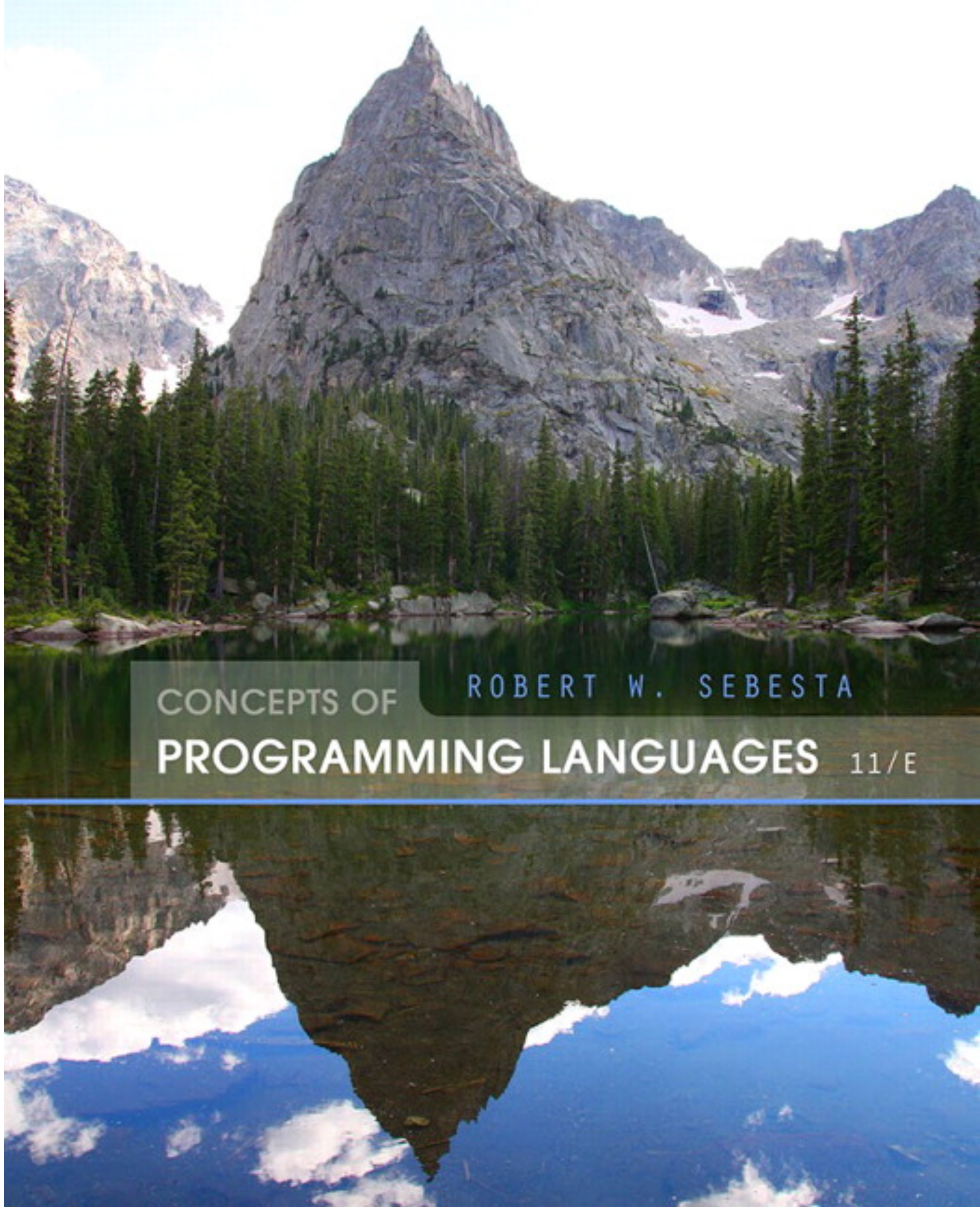
Chapter 3

Describing Syntax and Semantics



ISBN 0-321-49362-1

Static Semantics

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes

- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
  - For each grammar symbol *x* there is a set *A(x)* of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define *synthesized attributes*
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, *for* $i <= j <= n$, define *inherited attributes*
- Initially, there are *intrinsic attributes* on the leaves

Attribute Grammars: An Example

- Syntax

```
<assign> -> <var> = <expr>
<expr> -> <var> + <var> | <var>
<var> A | B | C
```

- `actual_type`: synthesized for `<var>` and `<expr>`
- `expected_type`: inherited for `<expr>`

Attribute Grammar (continued)

- **Syntax rule:** <expr> → <var>[1] + <var>[2]

  Semantic rules:

  <expr>.actual_type ← <var>[1].actual_type

  Predicate:

  <var>[1].actual_type == <var>[2].actual_type

  <expr>.expected_type == <expr>.actual_type

- **Syntax rule:** <var> → id

  Semantic rule:

  <var>.actual_type ← lookup (<var>.string)

Attribute Grammars (continued)

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

Attribute Grammars (continued)

```
<expr>.expected_type ← inherited from parent

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type
```

Semantics (dynamic semantics)

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for semantics:
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Designers could detect ambiguities and inconsistencies

Operational Semantics

- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed

Operational Semantics

- A *hardware* pure interpreter would be too expensive
- A *software* pure interpreter also has problems
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine-dependent

Operational Semantics (continued)

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Operational Semantics (continued)

- Uses of operational semantics:
  - Language manuals and textbooks
  - Teaching programming languages

- Two different levels of uses of operational semantics:
  - Natural operational semantics
  - Structural operational semantics

- Evaluation
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g.,VDL)

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called *assertions*

Axiomatic Semantics (continued)

- An assertion before a statement (a *precondition*) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a *postcondition*
- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition

Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`
- An example
  - `a = b + 1  {a > 1}`
  - One possible precondition: `{b > 10}`
  - Weakest precondition: `{b > 0}`

Program Proof Process

- The postcondition for the entire program is the desired result
  - Work back through the program to the first statement. If the precondition on the first statement is the same as the program specification, the program is correct.

Axiomatic Semantics: Assignment

- An axiom for assignment statements
  (x = E): $\{Q_{x->E}\}$ x = E $\{Q\}$

- The Rule of Consequence:

$$\frac{\{P\}\,S\,\{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\,S\,\{Q'\}}$$

Axiomatic Semantics: Sequences

- An inference rule for sequences of the form S1; S2

{P1} S1 {P2}

{P2} S2 {P3}

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

Axiomatic Semantics: Selection

- An inference rules for selection
  - **`if`** B **`then`** S1 **`else`** S2

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}}$$

Axiomatic Semantics: Loops

- An inference rule for logical pretest loops

{P} **`while`** B **`do`** S **`end`** {Q}

$$\frac{(\text{I and B})\ \text{S}\ \{\text{I}\}}{\{\text{I}\}\ \text{while B do S}\ \{\text{I and (not B)}\}}$$

where I is the loop invariant (the inductive hypothesis)

Evaluation of Axiomatic Semantics

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotation Semantics vs Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational