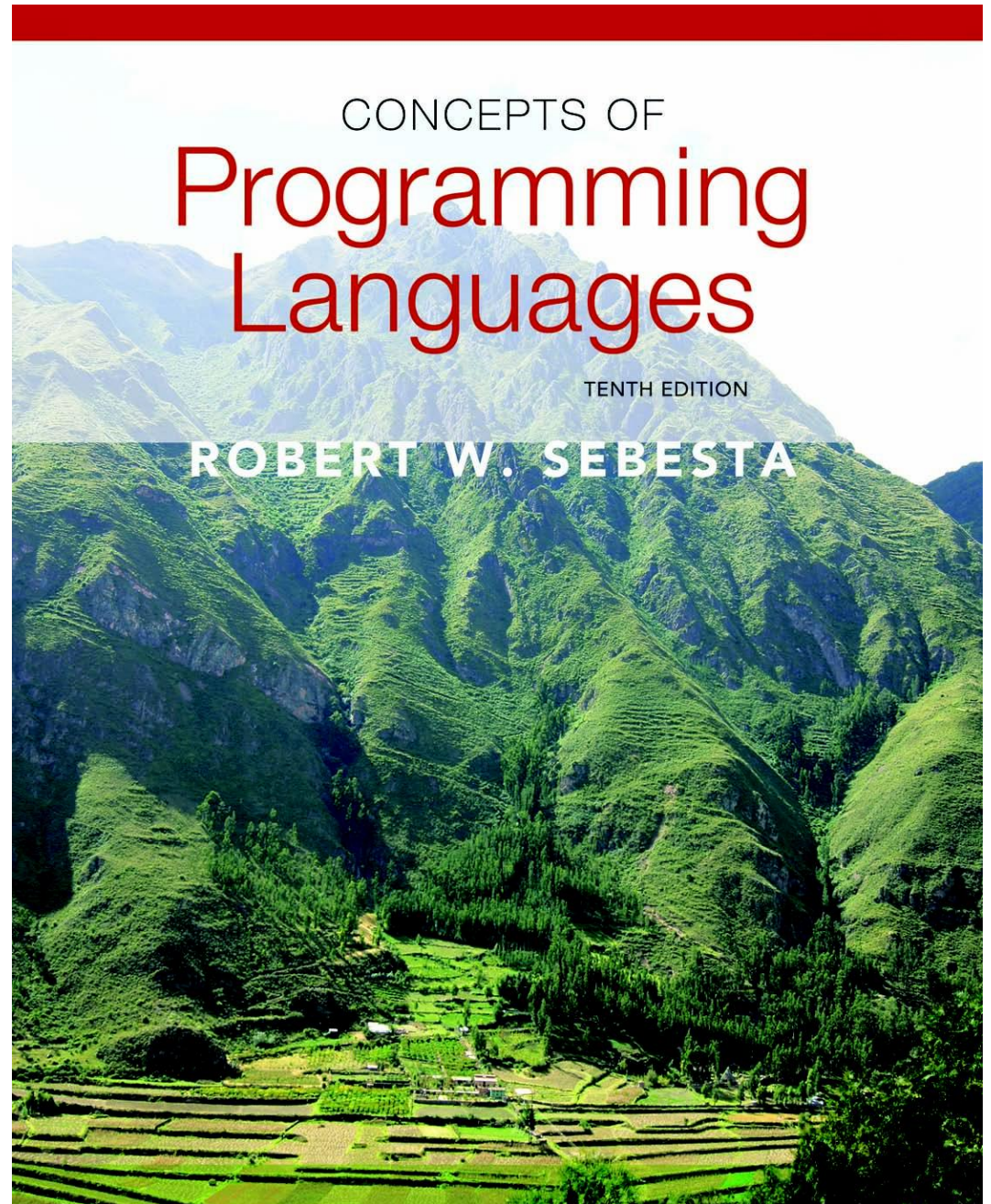


# Chapter 3

## Describing Syntax and Semantics



# Attribute Grammars

---

- Attribute grammars (AGs)
  - Knuth – 1968
- What is the difference?
  - AG vs CFG

# Attribute Grammars

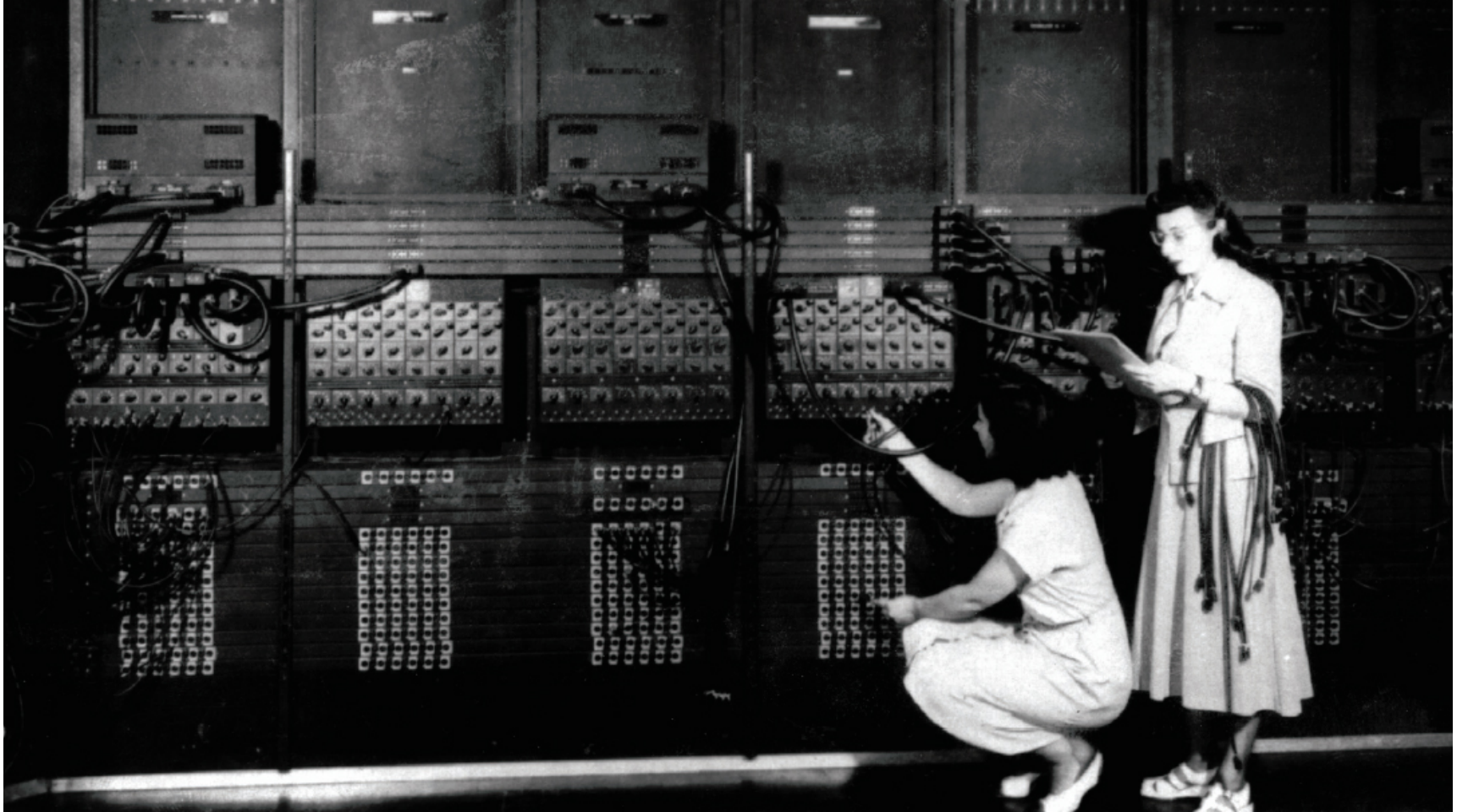
---

- Type Compatibility rules
  - Can you define them in CFG?
- Example:
  - In Java
    - float value = integer value
    - Integer value = float value ?
  - General
    - Variables should be declared before they are referenced.
      - Impossible to specify in BNF!



# ENIAC: Two Programmers

---



# Attribute Grammars

---

- Primary value of AGs:
  - Static semantics specification
    - Is related to the syntax (not semantics)!
  - Static semantics checking
    - (Compiler design)

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

# Attribute Grammars: An Example

---

- **Syntax**

$\langle \text{assign} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle \mid \langle \text{var} \rangle$

$\langle \text{var} \rangle A \mid B \mid C$

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



# Attribute Grammar (continued)

---

- **Syntax rule:**  $\langle \text{expr} \rangle \rightarrow \langle \text{var} \rangle[1] + \langle \text{var} \rangle[2]$

**Semantic rules:**

$\langle \text{expr} \rangle.\text{actual\_type} \leftarrow \langle \text{var} \rangle[1].\text{actual\_type}$

**Predicate:**

$\langle \text{var} \rangle[1].\text{actual\_type} == \langle \text{var} \rangle[2].\text{actual\_type}$

$\langle \text{expr} \rangle.\text{expected\_type} == \langle \text{expr} \rangle.\text{actual\_type}$

- **Syntax rule:**  $\langle \text{var} \rangle \rightarrow \text{id}$

**Semantic rule:**

$\langle \text{var} \rangle.\text{actual\_type} \leftarrow \text{lookup} (\langle \text{var} \rangle.\text{string})$

# 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

---

- 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
  - Compiler generators 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)

# Denotational Semantics

---

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

# Denotational Semantics – continued

---

- The process of building a denotational specification for a language:
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables

# Denotational Semantics: program state

---

- The state of a program is the values of all its current variables

$$s = \{ \langle i_1, v_1 \rangle, \langle i_2, v_2 \rangle, \dots, \langle i_n, v_n \rangle \}$$

- Let **VARMAP** be a function that, when given a variable name and a state, returns the current value of the variable

$$\text{VARMAP}(i_j, s) = v_j$$

# Decimal Numbers

---

`<dec_num>`  $\rightarrow$  '0' | '1' | '2' | '3' | '4' | '5' |  
'6' | '7' | '8' | '9' |  
`<dec_num>` ('0' | '1' | '2' | '3' |  
'4' | '5' | '6' | '7' |  
'8' | '9')

$M_{\text{dec}}('0') = 0, \quad M_{\text{dec}}('1') = 1, \quad \dots, \quad M_{\text{dec}}('9') = 9$

$M_{\text{dec}}(\text{<dec\_num> } '0') = 10 * M_{\text{dec}}(\text{<dec\_num>})$

$M_{\text{dec}}(\text{<dec\_num> } '1') = 10 * M_{\text{dec}}(\text{<dec\_num>}) + 1$

...

$M_{\text{dec}}(\text{<dec\_num> } '9') = 10 * M_{\text{dec}}(\text{<dec\_num>}) + 9$



# Expressions

---

- Map expressions onto  $\mathbb{Z} \cup \{\text{error}\}$
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

# Expressions

---

```
Me(<expr>, s) Δ=
  case <expr> of
    <dec_num> => Mdec(<dec_num>, s)
    <var> =>
      if VARMAP(<var>, s) == undef
        then error
      else VARMAP(<var>, s)
    <binary_expr> =>
      if (Me(<binary_expr>.<left_expr>, s) == undef
        OR Me(<binary_expr>.<right_expr>, s) =
          undef)
        then error
      else
        if (<binary_expr>.<operator> == '+' then
          Me(<binary_expr>.<left_expr>, s) +
            Me(<binary_expr>.<right_expr>, s)
        else Me(<binary_expr>.<left_expr>, s) *
          Me(<binary_expr>.<right_expr>, s)
    ...
```

# Assignment Statements

---

- Maps state sets to state sets  $U \{\text{error}\}$

```
Ma(x := E, s) Δ=
  if Me(E, s) == error
    then error
    else s' =
      {<i1, v1'>, <i2, v2'>, ..., <in, vn'>},
      where for j = 1, 2, ..., n,
        if ij == x
          then vj' = Me(E, s)
          else vj' = VARMAP(ij, s)
```

# Logical Pretest Loops

---

- Maps state sets to state sets  $U \{\text{error}\}$

```
M1(while B do L, s) Δ=  
  if Mb(B, s) == undef  
    then error  
  else if Mb(B, s) == false  
    then s  
  else if Ms1(L, s) == error  
    then error  
  else M1(while B do L, Ms1(L, s))
```

# Loop Meaning

---

- The meaning of the loop is the value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors
- In essence, the loop has been converted from iteration to recursion, where the recursive control is mathematically defined by other recursive state mapping functions
  - Recursion, when compared to iteration, is easier to describe with mathematical rigor

# Evaluation of Denotational Semantics

---

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, it are of little use to language users



# 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
  - $a = b + 1 \quad \{a > 1\}$
  - 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  
 $(x = E): \{Q_{x \rightarrow E}\} \ x = E \ \{Q\}$
- 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**

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

# Axiomatic Semantics: Loops

---

- An inference rule for logical pretest loops

$\{P\} \text{ while } B \text{ do } S \text{ end } \{Q\}$

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

where  $I$  is the loop invariant (the inductive hypothesis)

# Axiomatic Semantics: Axioms

---

- Characteristics of the loop invariant: I must meet the following conditions:
  - $P \Rightarrow I$       -- the loop invariant must be true initially
  - $\{I\} B \{I\}$       -- evaluation of the Boolean must not change the validity of I
  - $\{I \text{ and } B\} S \{I\}$       -- I is not changed by executing the body of the loop
  - $(I \text{ and } (\text{not } B)) \Rightarrow Q$       -- if I is true and B is false, Q is implied
  - The loop terminates      -- can be difficult to prove

# Loop Invariant

---

- The loop invariant  $I$  is a weakened version of the loop postcondition, and it is also a precondition.
- $I$  must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition

# 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**