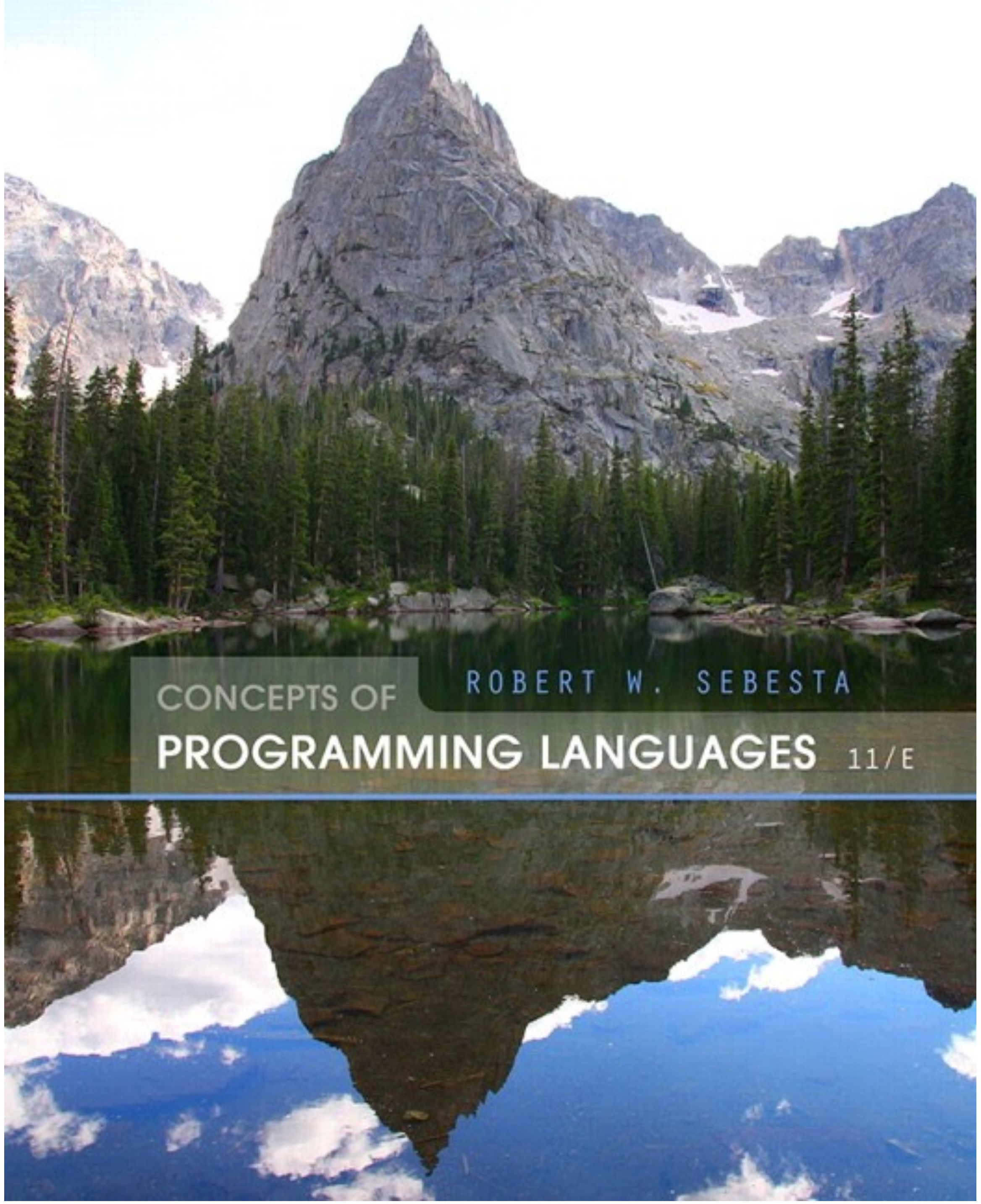
Chapter 3

Describing Syntax and Semantics



ISBN 0-321-49362-1

Chapter 3 Topics

- Describing the Meanings of Programs: Dynamic Semantics

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

Three Major Classes of Approaches

- Operational Semantics
  - the execution of the language is described directly (rather than by translation) ==> Interpretation.
  - Define an abstract machine, then give meaning to the execution of statements **(entire machine states' transitions)**
- Denotational Semantics
  - describing the meaning mathematically using recursive functions (**State of variables only)**
- Axiomatic Semantics
  - Define the meaning of statements by describing the logical axioms that apply to them: predicate calculus to prove the correctness (**State of relevant variables only)**

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 idealised computer)
  - Build a simulator for the idealised 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)

C Statement

for(expr1; expr2; expr3) { . . .

}

Meaning

expr1;
loop: **if** expr2 == 0 **goto** out

... expr3; **goto** loop

out: . . .

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

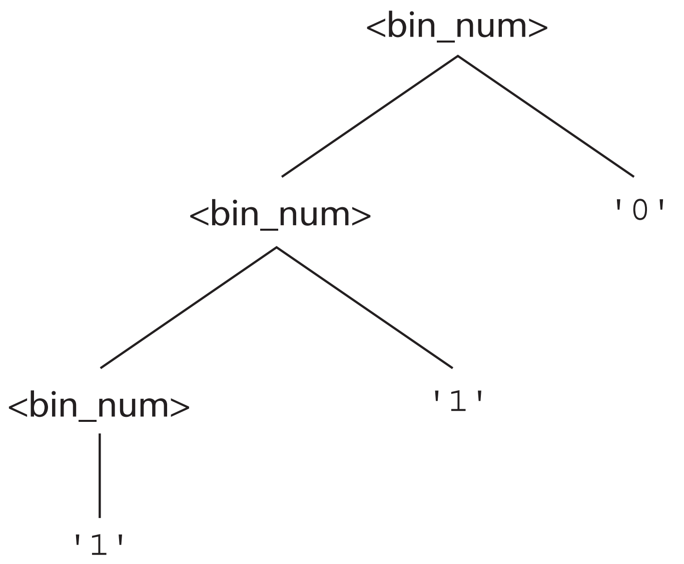
  $s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$

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

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

A parse tree for the example binary number, `110`

```
<bin_num> → '0'  | '1'
              | <bin_num> '0'
              | <bin_num> '1'
```

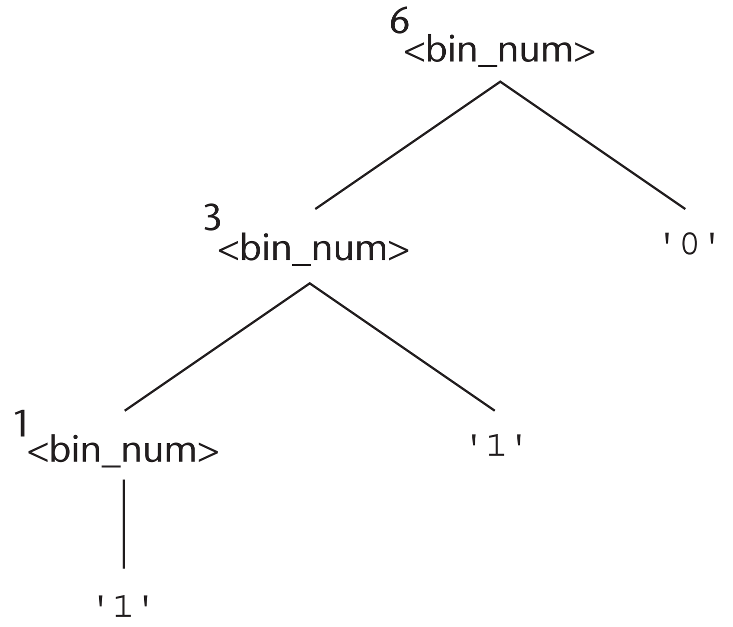


M_{bin}`('0') = 0`

M_{bin}`('1') = 1`

M_{bin}`(<bin_num> '0') = 2 * `M_{bin}`(<bin_num>)`

M_{bin}`(<bin_num> '1') = 2 * `M_{bin}`(<bin_num>) + 1`



Decimal Numbers

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

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Expressions

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

```
<expr> → <dec_num> | <var> | <binary_expr>
<binary_expr> → <left_expr> <operator> <right_expr>
<left_expr> → <dec_num> | <var>
<right_expr> → <dec_num> | <var>
<operator> → + | *
```

Expressions

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

Assignment Statements

- Maps state sets to state sets U {error}

```
$M_a$(x := E, s) Δ=
     if $M_e$(E, s) == error
        then error
        else s' =
   {<$i_1$,$v_1$'>,<$i_2$,$v_2$'>,...,<$i_n$,$v_n$'>},
                 where for j = 1, 2, ..., n,
                     if $i_j$ == x
                       then $v_j$' = $M_e$(E, s)
                       else $v_j$' = VARMAP($i_j$, s)
```

Logical Pretest Loops

- Maps state sets to state sets U {error}

```
M_l(while B do L, s) Δ=
    if M_b(B, s) == undef
        then error
        else if M_b(B, s) == false
            then s
            else if M_sl(L, s) == error
                then error
                else M_l(while B do L, M_sl(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 rigour

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 is 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}`
- `{P} S {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.

Inference Rules

$$\frac{S1, S2, \ \ldots\ , Sn}{S}$$

- This rule states that if S1, S2, . . . , and Sn are true, then the truth of S can be inferred.
- The top part of an inference rule is called its antecedent; the bottom part is called its consequent.
- An axiom is a logical statement that is assumed to be true. Therefore, an axiom is an inference rule without an antecedent.

Axiomatic Semantics: Assignment

- An axiom for assignment statements:

$P = Q_{x \to E}$

The Precondition P is computed as Q with all instances of x replaced by E.

- $(x = E)$: $\{Q_{x\text{->}E}\}$ $x = E$ $\{Q\}$
- Example:

`a = b / 2 - 1 {a < 10}`

the weakest precondition is computed by substituting `b / 2 - 1` for `a` in the postcondition `{a < 10}`, as follows:

`b / 2 - 1 < 10`

`b < 22`

The Rule of Consequence

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

P’ ==> P, means that P’ implies the assertion P,

In other words, the rule of consequence says that a postcondition can always be weakened and a precondition can always be strengthened.

ex: x = x - 3 {x > 0}, you can infer the precondition to be: {x > 3},

if it turns out to be: {x > 5} x = x - 3 {x > 0}, its alright.

Axiomatic Semantics: Sequences

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

{P1} S1 {P2}

{P2} S2 {P3}

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

```
ex:
S1 => y = 3 * x + 1;
S2 => x = y + 3;{x < 10}

therefore {Y < 7} is precondition to S2,
and postcondition to S1.

therefore {x < 2} is precondition to S1.

S1 => {x < 2} y = 3 * x + 1; {Y < 7}
S2 => {Y < 7} x = y + 3;     {x < 10}
```

Axiomatic Semantics: Selection

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

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

```
ex:
if x > 0 then y=y- 1

else y=y+ 1

{y > 0}

therefore {Y > 1} is precondition to then
clause, and {y > -1} to the else clause
```

because `{y > 1}` => `{y > -1}`, the rule of consequence allows us to use `{y > 1}` for the precondition of the whole selection statement.

Axiomatic Semantics: Loops

- An inference rule for logical pretest loops

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

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

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

Axiomatic Semantics: Axioms

- Characteristics of the loop invariant: I must meet the following conditions:
  - P => I -- the loop invariant must be true initially
  - {I} B {I} -- evaluation of the Boolean must not change the validity of I
  - {I and B} S {I} -- I is not changed by executing the body of the loop
  - (I and (not B)) => 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

Example:

» To find I, the loop postcondition Q is used to compute preconditions for several different numbers of iterations of the loop body, starting with none.

wp(statement, postcondition) = precondition

A wp function is often called a **predicate transformer**,

Ex:

```
while y <> x do y = y + 1 end {y = x}
```

For zero iterations, the weakest precondition is, obviously, : {y = x}
For one iteration, it is: wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x - 1}
For two iterations, it is: wp(y = y + 1, {y = x - 1})={y + 1 = x - 1}, or {y = x - 2}
For three iterations, it is: wp(y = y + 1, {y = x - 2})={y + 1 = x - 2}, or {y = x – 3}
It is now obvious that `{y <= x}` will suffice for cases of zero or more iterations, and can be used as loop invariant.

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