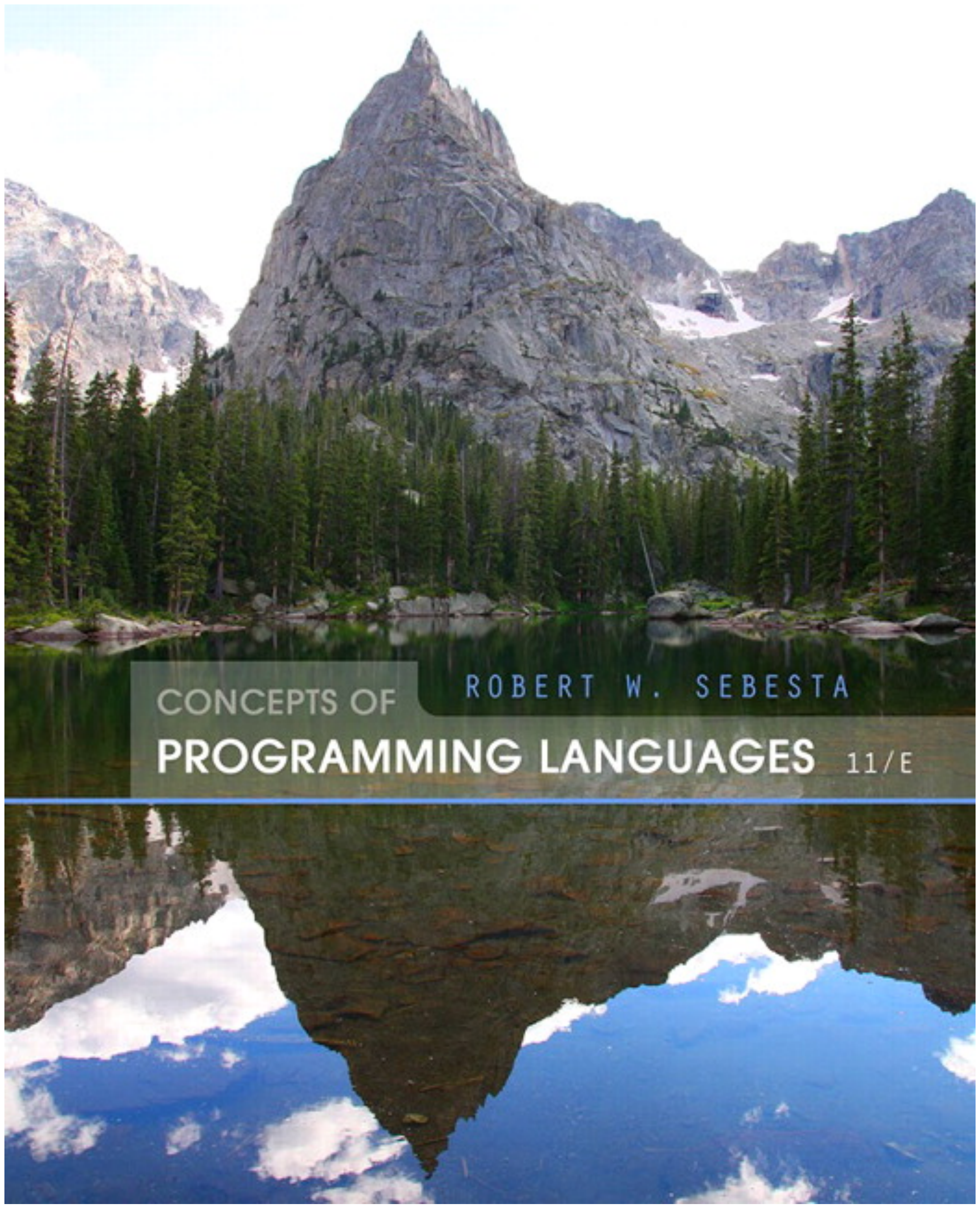
Chapter 3 (part 3)

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

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; Java floating-point value cannot be assigned to integer type, but opposite legal)
  - Non-context-free (e.g., variables must be declared before they are used)
- These type of needed specification checks are referred to as **Static Semantics**

Attribute Grammars

- Attribute grammars are used to describe more of the structure of PL than we can do with CFG, e.g. to address static semantics such as type compatibility
- 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 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, which state the static semantic rules, to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Synthesized attributes - up the parse tree from children
- Inherited attributes - down and across parse tree
- Initially, there are intrinsic attributes on the leaves (such as actual types of variables, int or real)

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: 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)

```
<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
```

Attribute Grammars (continued)

- checking static semantic rules of a language is an essential part of all compilers.

- main problems of contemporary languages?

Attribute Grammars (continued)

- checking static semantic rules of a language is an essential part of all compilers.

- main problems of contemporary languages?

  size and complexity

Semantics

- There is no single widely acceptable notation or formalism for describing semantics (or dynamic semantics; meaning while executing program)
- 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

Operational Semantics

- Design an appropriate intermediate level language that describes the high level language, where the primary goal is clarity
- A virtual machine (interpreter) must be constructed for the intermediate language
- Basic idea of translating to simpler, intermediate form, often used in textbooks or manuals to explain a language

Operational Semantics

Example: C for statement:

```
for (expr1; expr2; expr 3) {…}
for (a=1; a<10; a++) {…}
```

- Intermediate/meaning:

```
        expr1;
loop: if expr2==10 goto out
         …
         expr3;
         goto loop
out: …
```

Operational Semantics (continued)

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

- Evaluation
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally
    (e.g., VDL=Vienna Definition Language, to describe PL/I semantics in 1972)
  - Operational semantics depend on programming languages of a lower level, not on mathematics

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification; rather than directly specifying meaning of program asks what can be proved
- 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
- Assertions are used in programming languages (eg, Java, C)

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

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 input specification of the program, the program is correct.

We'll go through some examples…

Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`

- An example
  - a = b + 1
  - Postcondition: {a > 1}
  - One possible precondition: {b > 10}
  - Weakest precondition?

Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`
- `Want to find precondition that makes postcondition true`
- An example
  - a = b + 1
  - Postcondition: {a > 1}
  - One possible precondition: {b > 10}
  - Weakest precondition: {b > 0}

Axiomatic Semantics Form

- Pre-, post form: `{P} statement {Q}`
- `Want to find precondition that makes postcondition true`
- An example
  - a = b + 1
  - Postcondition: {a > 1}
  - One possible precondition: {b > 10}
  - Weakest precondition: {b > 0}
  - Written in pre-post form: {b>0} a = b + 1 {a>1}

Axiomatic Semantics Form

More examples on board…

Axiomatic Semantics: Axioms

- The Rule of Consequence:

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

Axiomatic Semantics: Axioms

- 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: Axioms

- While loop

  {P} while B do S end {Q}

  - If we knew how many iterations through the loop we could write out as sequence and prove correctness as before for the 2 sequence example, but we don't…
  - This is where the loop invariant (induction) comes to play

Axiomatic Semantics: Axioms

- Recall from CSC317…
- **Invariant** = something that does not change
- **Loop invariant** = a property about the algorithm that does not change at every iteration before the loop (it must be implied by precondition and thus hold before loop, and imply postcondition and thus hold after loop)
- This is usually the property we would like to prove is correct about the algorithm!

Axiomatic Semantics: Axioms

- Example with assert statements: http://www.cs.miami.edu/home/burt/learning/Csc517.101/workbook/findmin.html
- Input: Array A
- Output: Find minimum value in array A[1…n]
- Loop invariant: min is the smallest element in A[1 … i-1] before each iteration of the loop
- Loop invariant asserted before loop (precondition), each time goes through loop iteration, and at end of loop (postcondition)
- Done through induction (initialization, maintenance, termination)

Loop Invariant

- 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
- Here we develop more formally how to find loop invariant using axiomatic semantics

Axiomatic Semantics: Axioms

- Characteristics of the loop invariant: I must meet the following conditions:
  Loop example: {P} while B do S end {Q}
  - P => I -- the loop invariant must be true initially
  - {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 sometimes be difficult to prove

Axiomatic Semantics: Axioms

Loop invariant: formal example on the board…

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
- Its usefulness in describing the meaning of a programming language is highly limited for language users or compiler writers

Denotational Semantics

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

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
- Called **denotational** because mathematical constructs denote the meaning of the corresponding entities

Denotation Semantics vs Operational Semantics

- Operational semantics: programming constructs translated into simpler programming language constructs
- Denotational semantics: programming constructs mapped into mathematical functions

Example: Binary Numbers

meaning = number represents

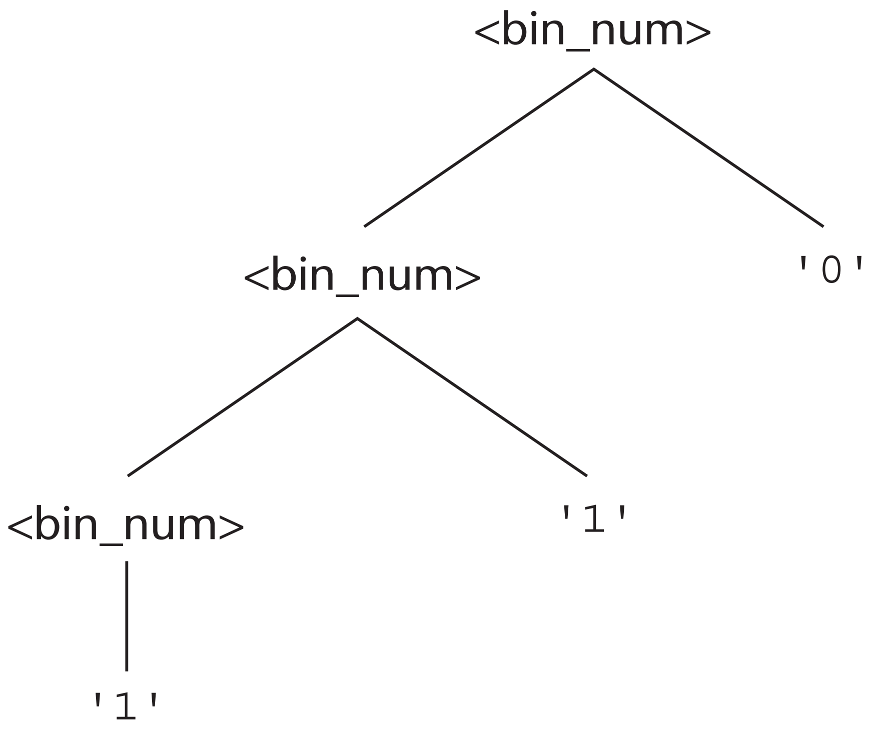
```
<bin_num> →  '0' | '1'
             |<bin_num> '0'
             |<bin_num> '1'
```

M_{bin}('0') = 0, M_{bin} ('1') = 1

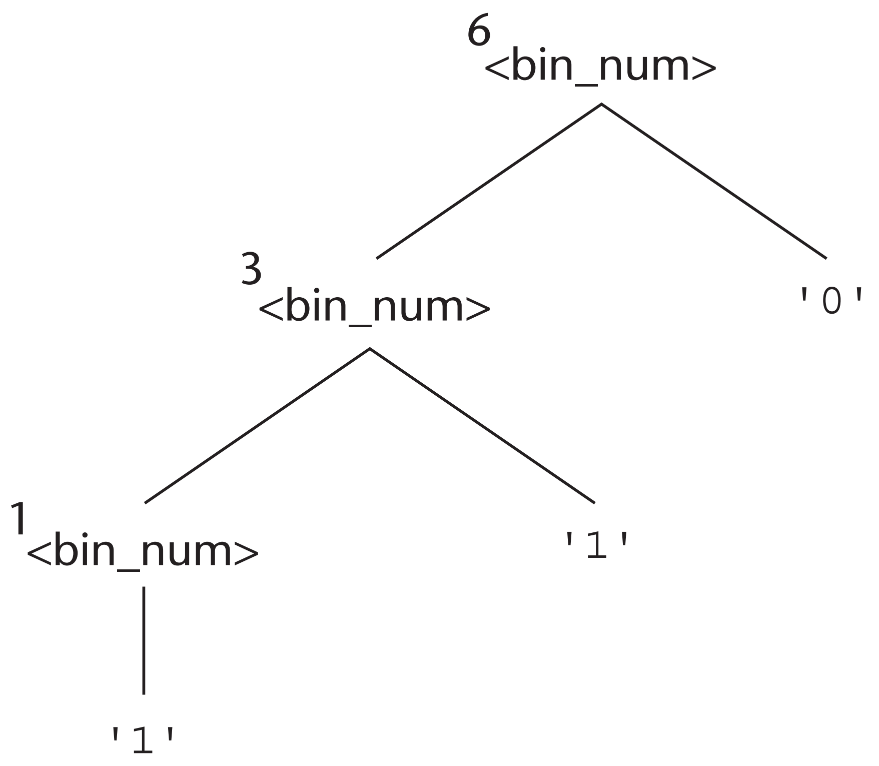
M_{bin} (<bin_num> '0') = 2 * M_{bin} (<bin_num>)

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

Example: Binary Numbers



Example: Binary Numbers



Numbers on nodes represent meaning

Denotational Semantics: Program State

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

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

```
i is name of variable and v
associated value
```

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

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

- Error we can consider is if VARMAP of a variable is undefined (special value **undef**)

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 - feasible but complicated!
- Because of its complexity, it’s of little use to language users

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