Chapter 3

Describing Syntax and Semantics

Chapter 3

Describing Syntax and Semantics

3.1 Introduction 110

- **Syntax** – the **form** of the expressions, statements, and program units
- **Semantics** - the **meaning** of the expressions, statements, and program units.
- Ex: the syntax of a Java while statement is

```
while (boolean_expr) statement
```

  - The semantics of this statement form is that when the current value of the Boolean expression is true, the embedded statement is executed.
  - The form of a statement should strongly suggest what the statement is meant to accomplish.

3.2 The General Problem of Describing Syntax 111

- A **sentence** or "**statement**" is a string of characters over some alphabet. The syntax rules of a language specify which strings of characters from the language's alphabet are in the language.
- A **language** is a set of sentences.
- A **lexeme** is the lowest level syntactic unit of a language. It includes identifiers, literals, operators, and special word (e.g. *, sum, begin). A **program** is strings of lexemes.
- A **token** is a category of lexemes (e.g., identifier). An identifier is a token that have lexemes, or instances, such as sum and total.
- Ex:

```
index = 2 * count + 17;
```

Lexemes	*Tokens*
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Language Recognizers and Generators

- In general, language can be formally defined in two distinct ways: by recognition and by generation.

- **Language Recognizers**:
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language.
  - It only determines whether given programs are in the language.
  - Example: syntax analyzer part of a compiler. The syntax analyzer, also known as **parsers**, determines whether the given programs are syntactically correct.

- **Language Generators**:
  - A device that generates **sentences** of a language
  - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

3.3 Formal Methods of Describing Syntax 113

- The formal language generation mechanisms are usually called **grammars**
- Grammars are commonly used to describe the **syntax** of programming languages.

3.3.1 Backus-Naur Form and Context-Free Grammars

- It is a syntax description formalism that became the **most widely** used method for programming language syntax.

3.3.1.1 Context-free Grammars

- Developed by Noam Chomsky in the mid-1950s who described four classes of generative devices or grammars that define four classes of languages.
- Context-free and regular grammars are useful for describing the syntax of programming languages.
- Tokens of programming languages can be described by regular grammars.
- Whole programming languages can be described by context-free grammars.

3.3.1.2 Backus-Naur Form (1959)

- Invented by John Backus to describe ALGOL 58 syntax.
- **BNF** (Backus-Naur Form) is equivalent to context-free grammars used for describing syntax.

3.3.1.3 Fundamentals

- A metalanguage is a language used to describe another language. **BNF** is a metalanguage for programming language.
- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called nonterminal symbols)

```
<assign> → <var> = <expression>
```

- This is a **rule**; it describes the structure of an assignment statement
- A rule has a left-hand side (**LHS**) "The abstraction being defined" and a right-hand side (**RHS**) "consists of some mixture of tokens, lexemes and references to other abstractions", and consists of terminal and nonterminal symbols.
- This particular rule specifies that the abstraction <assign> is defined as an instance of the abstraction <var>, followed by the lexeme =, followed by an instance of the abstraction <expression>.
- Example:

```
total = subtotal1 + subtotal2
```

- A grammar is a finite nonempty set of rules and the abstractions are called nonterminal symbols, or simply **nonterminals**.
- The lexemes and tokens of the rules are called terminal symbols or **terminals**.

- A BNF description, or grammar, is simply **a collection of rules**.
- An abstraction (or nonterminal symbol) can have more than one RHS
- For Example, a Java if statement can be described with the rule

```
<if_stmt> → if (<logic_expr>) <stmt>
          | if (<logic_expr>) <stmt> else <stmt>
```

- Multiple definitions can be written as a single rule, with the different definitions separated by the symbol |, meaning logical **OR**.

3.3.1.4 Describing Lists

- Syntactic lists are described using recursion.

```
<ident_list> → identifier
             | identifier, <ident_list>
```

- A rule is **recursive** if its LHS appears in its RHS.

3.3.1.5 Grammars and derivations

- The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the **start symbol**.
- A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
- Example 3.1 An Grammar for a 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>
```

- An example derivation for a program **begin A = B + C; B = C end**

```
<program>  => begin <stmt_list> end
           => begin <stm>; <stmt_list> end
           => begin <var> = <epression>; <stmt_list> end
           => begin A = <epression>; <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
```

- Every string of symbols in the derivation, including <program>, is 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. The derivation continues until the sentential form contains no nonterminals.
- A derivation may be neither leftmost nor rightmost.
- Example 3.2 An Grammar for a Small Assignment Statements:

```
<assign>  → <id> = <expr>
<id>      → A | B | C
<expr>    → <id> + <expr>
          | <id> * <expr>
          | (<expr>)
          | <id>
```

- An example derivation for the assignment **A = B * (A + C)**

```
<assign>  => <id> = <expr>
          => A = <expr>
          => A = <id> * <expr>
          => A = B * <expr>
          => A = B * (<expr>)
          => A = B * (<id> + <expr>)
          => A = B * (A + <expr>)
          => A = B * (A + <id>)
          => A = B * (A + C)
```

3.3.1.6 Parse Trees

- Hierarchical structures of the language are called **parse trees**.
- A parse tree for the simple statement `A = B + (A + C)`

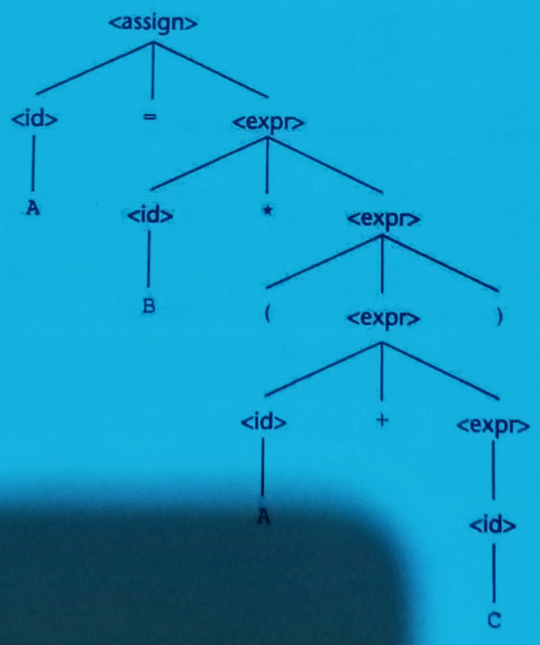
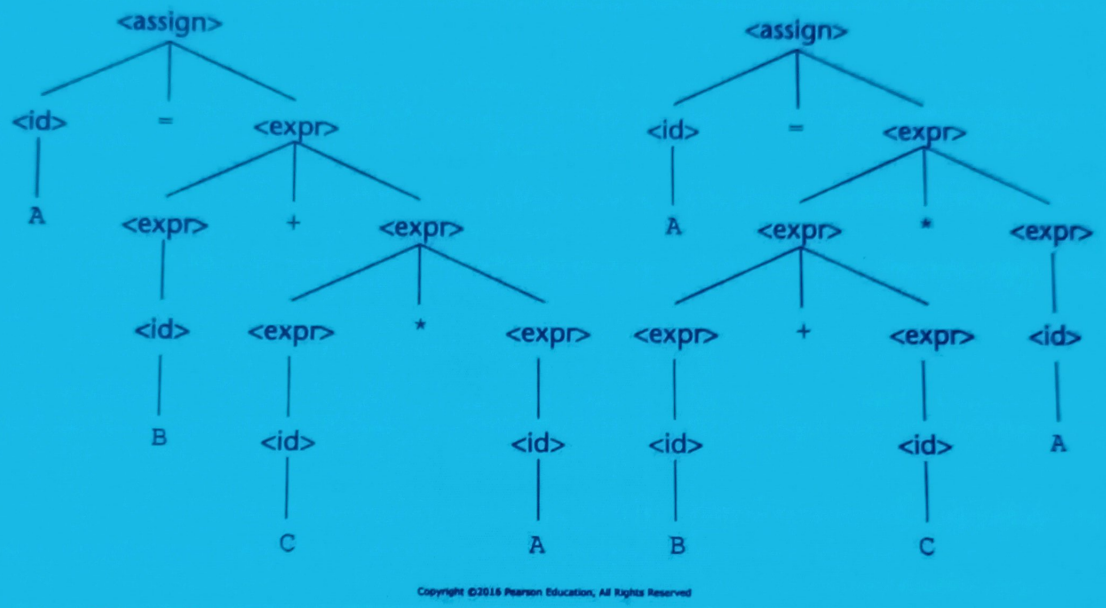


Figure 3.1 A parse tree for the simple statement A = B * (A + C)

3.3.1.7 Ambiguity

- A grammar is ambiguous if it generates a sentential form that has two or more distinct parse trees.
- Example 3.3 An Ambiguous Grammar for Small Assignment Statements
- Two distinct parse trees for the same sentence, **A = B + C * A**

```
<assign>  → <id> = <expr>
<id>      → A | B | C
<expr>    → <expr> + <expr>
          | <expr> * <expr>
          | (<expr>)
          | <id>
```



Copyright ©2016 Pearson Education, All Rights Reserved

Figure 3.2 Two distinct parse trees for the same sentence, A = B + C * A

3.3.1.8 Operator Precedence

- The fact that an operator in an arithmetic expression is generated **lower** in the parse tree can be used to indicate that it has **higher precedence** over an operator produced higher up in the tree.
- In the left parsed tree above, one can conclude that the * operator has precedence over the + operator. How about the tree on the right hand side?

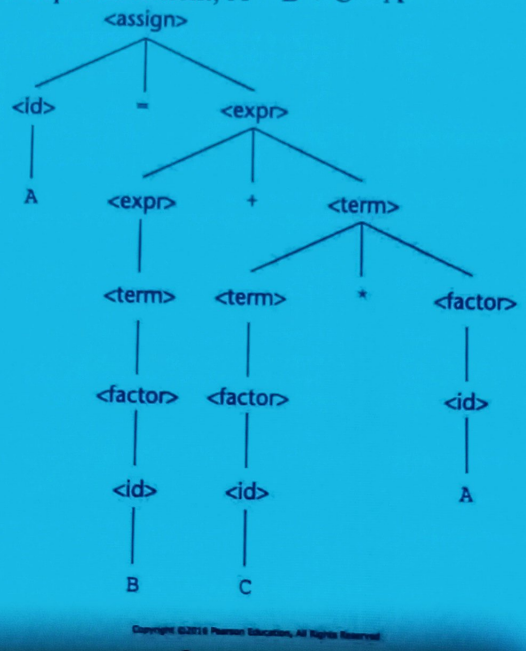
- Example 3.4 An **unambiguous** Grammar for Expressions

```
<assign>  → <id> = <expr>
<id>      →   A | B | C
<expr>    → <expr> + <term>
          | <term>
<term>    → <term> * <factor>
          | <factor>
<factor>  → (<expr>)
          | <id>
```

- **Leftmost derivation** of the sentence **A = B + C * A**

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <term>
         => A = <term> + <term>
         => A = <factor> + <term>
         => A = <id> + <term>
         => A = B + <term>
         => A = B + <term> * <factor>
         => A = B + <factor> * <factor>
         => A = B + <id> * <factor>
         => A = B + C * <factor>
         => A = B + C * <id>
         => A = B + C * A
```

A parse tree for the simple statement, A = B + C * A



Copyright ©2016 Pearson Education, All Rights Reserved

Figure 3.3 The unique parse tree for A = B + C * A using an unambiguous grammar

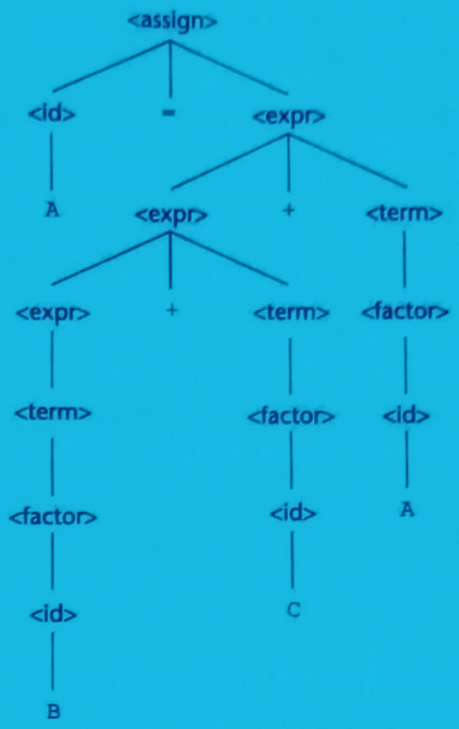
- **Rightmost derivation** of the sentence A = B + C * A

```
<assign>  => <id> = <expr>
          => <id> = <expr> + <term>
          => <id> = <expr> + <term> * <factor>
          => <id> = <expr> + <term> * <id>
          => <id> = <expr> + <term> * A
          => <id> = <expr> + <factor> * A
          => <id> = <expr> + <id> * A
          => <id> = <expr> + C * A
          => <id> = <term> + C * A
          => <id> = <factor> + C * A
          => <id> = <id> + C * A
          => <id> = B + C * A
          => A = B + C * A
```

- **Both** of these derivations, however, are represented by the **same** parse tree.

3.3.1.9 Associativity of Operators

- Do parse trees for expressions with two or more adjacent occurrences of operators with **equal precedence** have those occurrences in proper hierarchical order?
- An example of an assignment using the previous grammar is: A = B + C + A



Copyright ©2016 Pearson Education, All Rights Reserved

Figure 3.4 A parse tree for A = B + C + A illustrating the associativity of addition

- Figure above shows the left + operator lower than the right + operator. This is the correct order if + operator meant to be **left associative**, which is typical.
- When a grammar rule has LHS also appearing at beginning of its RHS, the rule is said to be left recursive. The left recursion specifies left associativity.
- In most languages that provide it, the **exponentiation operator** is **right associative**. To indicate right associativity, right recursion can be used. A grammar rule is **right recursive** if the LHS appears at the right **end** of the RHS. Rules such as

```
<factor>  → <exp> ** <factor>
          | <exp>
<exp>     → (<exp>)
          | id
```

3.3.2 Extended BNF

- Because of minor inconveniences in BNF, it has been extended in several ways. EBNF extensions do not enhance the descriptive powers of BNF; they **only** increase its readability and writability.
- Three extension are commonly included in various versions of EBNF
  - **Optional** parts are placed in brackets ([ ])

```
<if_stmt > → if (expression>) <statement> [else <statement>]
```

    - Without the use the brackets, the syntactic description of this statement would require the following two rules:

```
<if_stmt > → if (expression>) <statement>
           | if (expression>) <statement> else <statement>
```

  - Put **repetitions** (0 or more) in braces ({ })

```
<ident_list> → <identifier> {, <identifier>}
```

  - Put **multiple-choice** options of RHSs in parentheses and separate them with vertical bars (|, OR operator)

```
<term> → <term> (* | / | %) <factor>
```

    - In BNF, a description of this <term> would require the following three rules:

```
<term> → <term> * <factor>
       | <term> / <factor>
       | <term> % <factor>
```

- Example 3.5 BNF and EBNF Versions of an Expression Grammar

BNF:

```
<expr>   → <expr> + <term>
         | <expr> - <term>
         | <term>
<term>   → <term> * <factor>
         | <term> / <factor>
         | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp>    → (<expr>)
         | id
```

EBNF:

```
<expr>   → <term> {(+ | -) <term>}
<term>   → <factor> {(* | /) <factor>}
<factor> → <exp> {** <exp>}
<exp>    → (<expr>)
         | id
```

3.4 Attribute Grammars 128

- An attribute grammar is a device used to describe **more** of the structure of a programming language than can be described with a context-free grammar.

3.4.1 Static Semantics

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages.
- In Java, for example, a floating-point value **cannot** be assigned to an integer type variable, although the opposite is legal.
- The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics).
- Many static semantic rules of a language state its type constraints. Static semantics is so named because the analysis required to these specifications can be done at **compile time**.
- **Attribute grammars** was designed by Knuth (1968) to describe both the **syntax** and the **static semantics** of programs.

3.4.2 Basic Concepts

- Attribute grammars have additions to are context-free grammars to carry some **semantic** information on parse tree nodes.
- Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate function.

3.4.3 Attribute Grammars Defined

- Associated with each grammar symbol X is a set of attributes A(X).
  - The set A(X) consists of two disjoint set S(X) and I(X), call synthesized and inherited attributes.
  - Synthesized attributes are used to pass semantic information **up** a parse tree, while inherited attributes pass semantic information **down** and across tree.
- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
  - Functions of the form $S(X_0) = f(A(X_1), . , A(X_n))$ 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

3.4.4 Intrinsic Attributes

- Intrinsic attributes are **synthesized** attributes of **leaf** nodes whose values are determined **outside** the parse tree.
- For example, the type of an instance of a variable in a program could come form the **symbol table**, which is used to store variable names and their types.

3.4.5 Examples Attribute Grammars

- Example 3.6 An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int and (<var>[2].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type

- The syntax portion of our example attribute grammar is

```
<assign> → <var> = <expr>
<expr>   → <var>[2] + <var>[3]
         | <var>
<var>    → A | B | C
```

- actual_type - A **synthesized** attribute associated with nonterminal <var> and <expr>.
  - It is used to store the actual type, int or real, or a variable or expression.
  - In the case of a variable, the actual type is intrinsic.
  - In the of an expression, it is determined from the actual types of child node or children nodes of the <expr> nonterminal.
- expected_type - A **inherited** attribute associated with nonterminal <expr>.
  - It is used to store the type, either int or real, that is expected for the expression, as determined by the type of the variable on the left side of the assignment statement.

3.4.6 Computing Attribute Values

- Now, consider the process of computing the attribute values of a parse tree, which is sometimes called decorating the parse tree.
- The tree in Figure 3.7 show the flow of attribute values in the example of Figure 3.6.

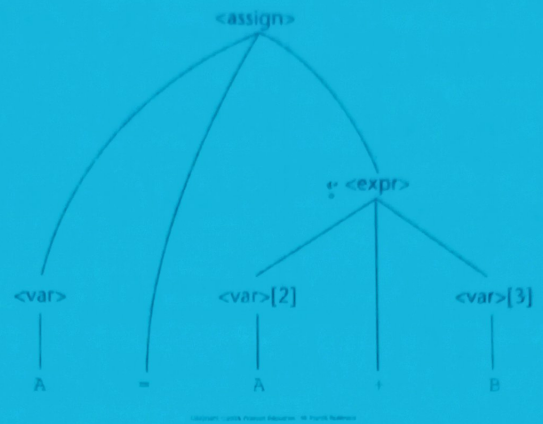


Figure 3.6 A parse tree for A = A + B

- The following is an evaluation of the attributes, in an order in which it is possible to computer them:

1.	<var>.actual_type	←	look-up (A)	(Rule 4)
2.	<expr>.expected_type	←	<var>.actual_type	(Rule 1)
3.	<var>[2].actual_type	←	look-up (A)	(Rule 4)
	<var>[3].actual_type	←	look-up (B)	(Rule 4)
4.	<expr>.actual_type	←	either int or real	(Rule 2)
5.	<expr>.expected_type	==	<expr>.actual_type is either TRUE or FALSE (Rule 2)	

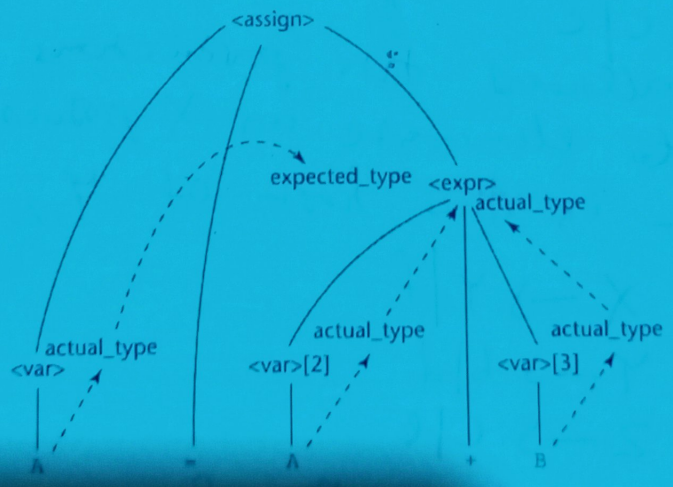


Figure 3.7 The flow of attributes in the tree

Tutorial Questions

1) Give a grammar with the following first and follow sets. Your grammar should have exactly two productions per non-terminal and no epsilon productions. The non-terminals are X, Y, Z and the terminals are a, b, c, d, e, f.

First(X) = {b, d, f}	Follow(X) = {$}
First(Y) = {b, d}	Follow(Y) = {c, e}
First(Z) = {c, e}	Follow(Z) = {a}
Follow(d) = {c, e}	Follow(a) = {$}
Follow(e) = {a}	Follow(b) = {b, d}
Follow(f) = {$}	Follow(c) = {c, e}

There are many ways to solve this problem. In our solution, we began by using the first sets to generate the obvious productions that have the correct first (and only) terminal:

X → b | d | f
Y → b | d
Z → c | e

We are only allowed two productions per non-terminal, so we need to eliminate an X production. Noticing that b and d are in the first of Y, we can try

X → Y | f
Y → b | d
Z → c | e

At this point we have the first sets correct for the non-terminals; now we try to modify the productions to have the correct follow sets as well. Starting arbitrarily with the terminal c, we notice that both c and e are in follow(C) and in first(Z), so we can add Z after C in some production. There is only one choice:

- The tree in Figure 3.8 shows the final attribute values on the nodes. In this example, A is defined as a real and B is defined as an int.

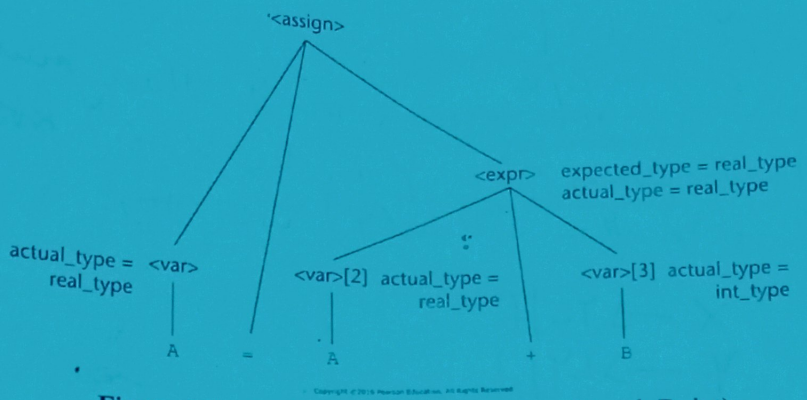


Figure 3.8 A fully attributed parse tree (A: real, B: int)

3.4.7 Evaluation

- Checking the static semantic rules of a language is an essential part of all **compiler**.
  - One of the main difficulties in using an attribute grammar to describe all of the syntax and static semantics of a real contemporary programming language is the **size and complexity** of the attribute grammar.
  - Furthermore, the attribute values on a large parse tree are **costly** to evaluate.

$X \to Y \mid f$
$Y \to b \mid d$
$Z \to cZ \mid e$

Similar reasoning about follow(b) leads to:

$X \to Y \mid f$
$Y \to bY \mid d$
$Z \to cZ \mid e$

Since $ is not in the follow of Y, there must be something that comes after Y in the $X \to Y$ production. Since first(Z) is in the follow(Y) we try

$X \to YZ \mid f$
$Y \to bY \mid d$
$Z \to cZ \mid e$

At this point we still haven't used a. Also follow(a) = {$} and follow(Z) = {a}. So we get

$X \to YZa \mid f$
$Y \to bY \mid d$
$Z \to cZ \mid e$

At this point, it is routine to check that all the requirements are satisfied.

(2) Consider the following grammar

$S \to Sa \mid b \mid error\ a$

Show the DFA for recognizing viable prefixes of this grammar. Use LR(0) items, and treat error as a terminal.

States

$I_0 = \{S \to \cdot Sa, S \to \cdot b, S \to \cdot error\ a\}$
$I_1 = \{S \to S \cdot a\}$
$I_2 = \{S \to b \cdot\}$
$I_3 = \{S \to error \cdot a\}$
$I_4 = \{S \to error\ a \cdot\}$
$I_5 = \{S \to Sa \cdot\}$

- Three methods of semantic description:
  - **Operational semantics:** It is a method of **describing** the meaning of language constructs in terms of their effects on an ideal machine.
  - **Denotation semantics: Mathematical** objects are used to represents the meanings of languages constructs. Language entities are converted to these mathematical objects with recursive functions.
  - **Axiomatic semantics:** It is based on formal **logic** and devised as a tool for proving the correctness of programs.

3.5.1 Operational Semantics

- The idea behind **operational semantics** is to describe the meaning of a statement or program, by specifying the effects of running it on a machine. The effects on the machine are viewed as the sequence of **changes in its states**, where the machine's state is the collection of the collection of the values in its storage.
- Most programmers have written a small **test** program to determine the meaning of some programming language construct.
- The basic process of operational semantics is not unusual. In fact, the concept is frequently used in programming textbooks and programming in reference manuals.
- For example, the semantics of the C **for** construct can described in terms of simpler statements, as in

C Statement

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

Meaning

```
      expr1;
loop: if expr2 == 0 goto out
      ...
      expr3;
      goto loop
out:  ...
```

- Operational semantics **depends** on programming languages of lower level, **not** mathematics and logic.

Transitions

$I_0 - s \rightarrow I_1$

$I_0 - b \rightarrow I_2$

$I_0 - error \rightarrow I_3$

$I_1 - a \rightarrow I_5$

$I_3 - a \rightarrow I_4$

---

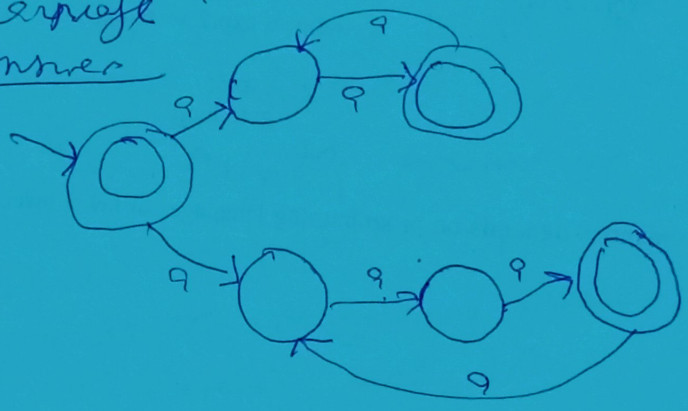
(3) Consider the language over the alphabet $\Sigma = \{a\}$ containing strings whose length is either a multiple of 2 or a multiple of 3 (this excludes the empty string). Write a regular expression for this language.

Answer

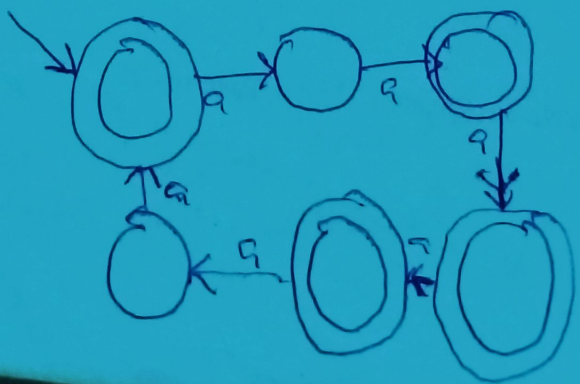
$$(aa)^* / (aaa)^*$$

(b) Draw a non-deterministic finite automaton (NFA) for this language.

Answer



(c) Draw a deterministic finite automaton (DFA) for this language



3.5.2 Denotational Semantics

- **Denotational semantics** is the most rigorous and most widely known formal method for describing the meaning of programs.
- It is solidly based on **recursive** function theory.

Two Simple Examples

- Example 1:
  - We use a very simple language construct, character string representations of **binary numbers**, to introduce the denotational method.
  - The syntax of such binary numbers can be described by the following grammar rules:

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

  - A parse tree for the example binary number, **110**, is show in Figure 3.9.

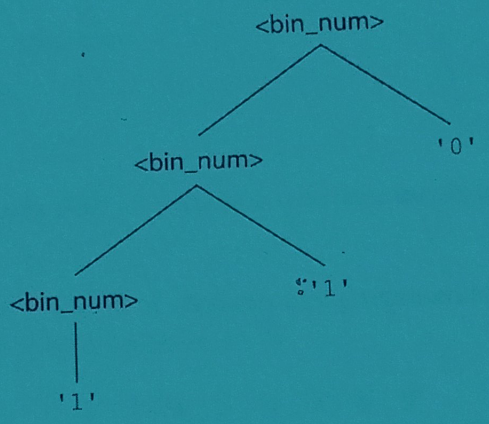


Figure 3.9 A parse tree of the binary number 110

- The semantic function, named M_{bin}, maps the syntactic objects, as described in the previous grammar rules, to the objects in N, the set of non-negative decimal numbers. The function M_{bin} is defined as follows:

```
Mbin ('0') = 0
Mbin ('1') = 1
Mbin (<bin_num> '0') = 2 * Mbin (<bin_num>)
Mbin (<bin_num> '1') = 2 * Mbin (<bin_num>) + 1
```

- The meanings, or denoted objects (which in this case are decimal numbers), can be attached to the nodes of the parse tree, yielding the tree in Figure 3.10)

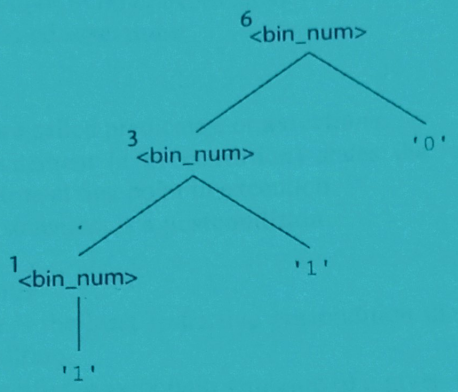


Figure 3.10 A parse tree with denoted objects for 110

- Example 2:
  - The syntactic domain is the set of character string representations of **decimal numbers**. The semantic domain is once again the set N.

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

  - The denotational mapping for these syntax rules are

M_{dec} ('0') = 0, M_{dec} ('1') = 1, M_{dec} ('2') = 2, …, 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

3.5.3 Axiomatic Semantics

- Axiomatic Semantics is based on **mathematical logic**.
- It is defined in conjunction with the development of a method to **prove** the correctness of programs.
  - Such correction proofs, when they can be constructed, show that a program performs the computation described by its specification.
  - In a proof, each statement of a program is both preceded and followed by a logical expression that specified constraints on program variables.
- Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions.)
  - The expressions are called **assertions**.

3.5.3.1 Assertions

- The logical expressions are called predicates, or **assertions**.
- 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**.

3.5.3.2 Weakest Preconditions

- A **weakest precondition** is the least restrictive precondition that will guarantee the validity of the associated postcondition.
- The usual notation for specify the axiomatic semantic of a given statement form is

  ```
  {P} statement {Q}
  ```

  where P is the precondition, Q is the postcondition, and S is the statement form

  - An example: a = b + 1 {a > 1}

    One possible precondition: {b > 10}
    Weakest precondition: **{b > 0}**

- If the weakest precondition can be computed from the given postcondition for each statement of a language, then correctness proofs can be constructed from programs in that language.
- **Program proof process**: The postcondition for the whole 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 spec, the program is correct.
- An **Axiom** is a logical statement that is assumed to be true.

- An **Inference Rule** is a method of inferring the truth of one assertion on the basis of the values of other assertions.

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

  - The 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 call its antecedent; the bottom part is called it consequent.

3.5.3.3 Assignment Statements

- Ex:

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

The weakest precondition is computed by substituting b / 2 -1 in the assertion {a < 10} as follows:

```
b / 2 - 1< 10
b / 2     < 11
b         < 22
```

∴ the weakest precondition for the given assignment and the postcondition is **{b < 22}**

- An assignment statement has a side effect if it changes some variable other than its left side.
- Ex:

```
x = 2 * y - 3 {x > 25}
```

The weakest precondition is computed as follows:

```
2 * y - 3> 25
2 * y     > 28
y         > 14
```

∴ the weakest precondition for the given assignment and the postcondition is **{y > 14}**

- Ex:

```
x = x + y - 3 {x > 10}
```

The weakest precondition is computed as follows:

```
x + y - 3   > 10
y           > 13 - x
```

∴ the weakest precondition for the given assignment and the postcondition is **{y > 13 -x}**

3.5.3.4 Sequences

- The weakest precondition for a sequence of statements cannot be described by an axiom, because the precondition depends on the particular kinds of statements in the sequence.
- In this case, the precondition can only be described with an inference rule.
- Let S1 and S2 be adjacent program statements. If S1 and S2 have the following preconditions and postconditions.

```
{P1} S1 {P2}
{P2} S2 {P3}
```

The inference rule for such a two-statement sequence is

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

- Ex:

```
y = 3 * x + 1;
x = y + 3;      {x < 10}
```

The weakest precondition is computed as follows:

```
y + 3  < 10
y      < 7

3 * x + 1   < 7
3 * x       < 6
x           < 2
```

∴ the weakest precondition for the first assignment statement is **{x < 2}**

3.5.3.5 Selection

- We next consider the inference rule for selection statements, the general form of which is

```
if B then S1 else S2
```

We consider only selections that include else clause. The inference rule is

```
{B and P} S1 {Q}, {not B and P} S2 {Q}
-----------------------------------------
      {P} if B then S1 else S2 {Q}
```

- Example of selection statement is

```
If (x > 0) then
     y = y - 1;
else
     y = y + 1;
{y > 0}
```

We can use the axiom for assignment on the then clause
$y = y - 1$ $\{y > 0\}$
This produce precondition $\{y - 1 > 0\}$ or $\{y > 1\}$

Now we apply the same axiom to the else clause
$y = y + 1$ $\{y > 0\}$
This produce precondition $\{y + 1 > 0\}$ or $\{y > -1\}$

$y > 1$ AND $y > -1$
$\{y > 1\}$
Because $\{y > 1\} \Rightarrow \{y > -1\}$, the rule of consequence allows us to use $\{y > 1\}$ for the precondition of selection statement.

3.5.3.6 Logical Pretest Loops

- Computing the weakest precondition (wp) for a while loop is inherently more difficult than for a sequence b/c the number of iterations cannot always be predetermined.
- The corresponding step in the axiomatic semantics of a **while** loop is finding an assertion called a **loop invariant**, which is crucial to finding the weakest precondition.
- The inference rule for computing the precondition for a while loop is as follows:

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

In this rule, I is the loop invariant

- The axiomatic description of a while loop written as

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

- It is helpful to treat the process of producing the wp as a function, **wp**.

```
wp(statement, postcondition) = precondition
```

- To find **I**, we use the loop postcondition to compute preconditions for several different numbers of iterations of the loop body, starting with none. If the loop body contains a single assignment statement, the axiom for assignment statements can be used to compute these cases.
- 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

- Ex:

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

For 0 iterations, the wp is → {y = x}
For 1 iteration,
wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x – 1}
For 2 iterations,
wp(y = y + 1, {y = x - 1}) = {y + 1 = x - 1}, or {y = x – 2}
For 3 iterations,
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 one or more iterations. Combining this with {y = x} for the 0 iterations case, we get **{y <= x}** which can be used for the loop invariant.

- Ex:

```
while s > 1 do s = s / 2 end {s = 1}
```

For 0 iterations, the wp is → {s = 1}
For 1 iteration,
wp(s > 1, {s = s / 2}) = {s / 2 = 1}, or {s = 2}
For 2 iterations,
wp(s > 1, {s = s / 2}) = {s / 2 = 2}, or {s = 4}
For 3 iterations,
wp(s > 1, {s = s / 2}) = {s / 2 = 4}, or {s = 8}

- Loop Invariant I is {s is a **nonnegative power of 2**}
- 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.

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, denotational, axiomatic**

Chapter 5

Names, Bindings, and Scopes

Chapter 5

Names, Bindings, and Scopes

5.1 Introduction 198

- Imperative languages are abstractions of von Neumann architecture
  - Memory: stores both instructions and data
  - Processor: provides operations for modifying the contents of memory
- Variables are characterized by a collection of properties or attributes
  - The most important of which is **type**, a fundamental concept in programming languages
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

5.2 Names 199

5.2.1 Design issues

- The following are the primary design issues for names:
  - Maximum length?
  - Are names case sensitive?
  - Are special words reserved words or keywords?

5.2.2 Name Forms

- A **name** is a string of characters used to identify some entity in a program.
- Length
  - If too short, they cannot be connotative
  - Language examples:
    - FORTRAN I: maximum 6
    - COBOL: maximum 30
    - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
    - C# and Java: **no limit**, and all characters are significant
    - C++: **no limit**, but implementers often impose a length limitation because they do not want the **symbol table** in which identifiers are stored during compilation to be too large and also to simplify the maintenance of that table.
- Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and (_). Although the use of the _ was widely used in the 70s and 80s, that practice is far less popular.
- **C-based** languages (C, Objective-C, C++, Java, and C#), replaced the _ by the "camel" notation, as in myStack.

- Prior to Fortran 90, the following two names are equivalent:

```
Sum Of Salaries     // names could have embedded spaces
SumOfSalaries       // which were ignored
```

- Special characters
  - PHP: all variable names must begin with dollar signs $
  - Perl: all variable names begin with special characters $, @, or %, which specify the variable's type
    - if a name begins with $ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure
  - Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables
- Case sensitivity
  - Disadvantage: readability (names that look alike are different)
  - Names in the C-based languages are case sensitive
  - Worse in C++, Java, and C# because predefined names are mixed case (e.g. IndexOutOfBoundsException)
  - In C, however, exclusive use of lowercase for names.
    - C, C++, and Java names are case sensitive ➔ rose, Rose, ROSE are distinct names "What about Readability"

5.2.3 Special words

- An aid to readability; used to delimit or separate statement clauses
- A **keyword** is a word that is special only in certain contexts.
- Ex: Fortran

```
Real Apple          // Real is a data type followed with a name,
                    therefore Real is a keyword
Real = 3.4          // Real is a variable name
```

- Disadvantage: poor readability. Compilers and users must recognize the difference.
- A **reserved** word is a special word that **cannot** be used as a user-defined name.
- Potential problem with reserved words: If there are too many, many collisions occur (e.g., COBOL has **300** reserved words!)
- As a language design choice, reserved words are **better** than keywords.
- Ex: In Fortran, they are **only** keywords, which means they can be redefined. One could have the statements:

```
Integer     Real        // keyword "Integer" and variable "Real"
Real        Integer     // keyword "Real" and variable "Integer"
```

5.3 Variables 200

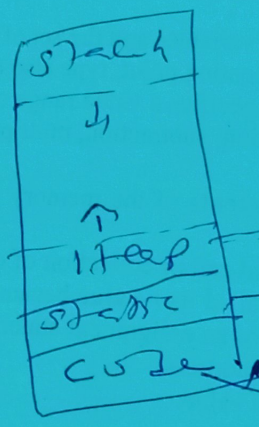
- A variable is an abstraction of a memory cell.
- Variables can be characterized as a sextuple of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

- Name
  - Not all variables have names: Anonymous, heap-dynamic variables
- Address
  - The memory address with which it is associated
  - A variable may have **different** addresses at **different** times during execution. If a subprogram has a local var that is allocated from the run time stack when the subprogram is called, different calls may result in that var having different addresses.
  - The address of a variable is sometimes called its ***l*-value** because that is what is required when a variable appears in the **left** side of an assignment statement.
- Aliases
  - If two variable names can be used to access **the same** memory location, they are called aliases
  - Aliases are created via **pointers**, **reference variables**, C and C++ **unions.**
  - Aliases are harmful to readability (program readers must remember all of them)
- Type
  - Determines the **range** of values of variables and the set of **operations** that are defined for values of that type; in the case of floating point, type also determines the precision.
  - For example, the int type in Java specifies a value range of -2147483648 to 2147483647, and arithmetic operations for addition, subtraction, multiplication, division, and modulus.
- Value
  - The value of a variable is the **contents** of the memory cell or cells associated with the variable.
  - Abstract memory cell - the physical cell or collection of cells associated with a variable.
  - A variable's value is sometimes called its ***r*-value** because that is what is required when a variable appears in the **right** side of an assignment statement.
    - The ***l*-value** of a variable is its address.
    - The ***r*-value** of a variable is its value.

- Aliases
- Ref. & use pointers to create aliases
- Scope of a variable
- Explicit & implicit declaration of variables - Examples
- Binding in programming languages Dynamic & static binding

```
int i, a[10]
void fn()
  { int i, j, *p;
    ...=j;
    i=...;
    p=malloc();
  }
```

→ explicitly creates space eg malloc()



explicitly creates space

Static or global variables, declared once per program

Program code

```
int i a[10];
void fn()
  { int i, j;
    ...=j;
    i=...;
  }
void bar()
  { int j;
    ...=j;
    i=...;
```

which address is variable i bound?

5.4 The Concept of Binding 203

- A **binding** is an association, such as between an attribute and an entity, or between an operation and a symbol.
- **Binding time** is the time at which a binding takes place.
- Possible binding times:
  - Language design time: bind operator symbols to operations.
    - For example, the asterisk symbol (*) is bound to the multiplication operation.
  - Language implementation time:
    - A data type such as **int** in C is bound to a **range** of possible values.
  - Compile time: bind a variable to a **particular data type** at compile time.
  - Load time: bind a variable to a **memory cell** (ex. C **static** variables)
  - Runtime: bind a **nonstatic** local variable to a memory cell.

5.4.1 Binding of Attributes to Variables

- A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
- A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.

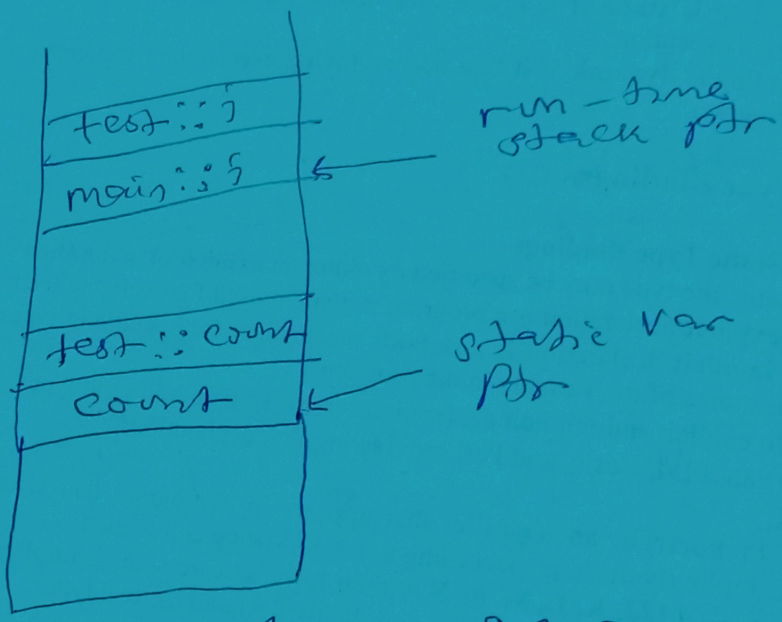
5.4.2 Type Bindings

5.4.2.1 Static Type Bindings

- If static, the type may be specified by either an **explicit** or an **implicit** declaration.
- An **explicit** declaration is a program statement used for declaring the types of variables.
- An **implicit** declaration is a **default** mechanism for specifying types of variables (the first appearance of the variable in the program.)
- Both explicit and implicit declarations create static bindings to types.
- Fortran, PL/I, Basic, and Perl provide implicit declarations.
- EX:
  - In **Fortran**, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following convention:
    I, J, K, L, M, or N or their lowercase versions is **implicitly** declared to be Integer type; otherwise, it is implicitly declared as Real type.
  - Advantage: writability.
  - Disadvantage: reliability suffers because they prevent the compilation process from detecting some typographical and programming errors.
  - In Fortran, vars that are accidentally left undeclared are given default types and unexpected attributes, which could cause subtle errors that, are difficult to diagnose.
  - Less trouble with **Perl**: Names that begin with $ is a scalar, if a name begins with @ it is an array, if it begins with %, it is a hash structure.
  - In this scenario, the names @apple and %apple are unrelated.

Stack — dynamic variables

```
#include <stdio.h>
int count;
main() {
    int i;
    for (i=0; i<=10; i++)
    { test(); }
}
test() {
    int i;
    static int count = 0;
    count = count + 1;
}
```



Virtual address space

Allocation & Deallocation of variable

```
Person *p;
p = (Person *) malloc (size of person);
p->name = "Mike"; p->age = 40;
free(p);
```

- **Type Inference:** Some languages use type inferencing to determine types of variables (context)
  - **C#** - a variable can be declared with **var** and an initial value. The initial value sets the type

```
var sum = 0;            // sum is int
var total = 0.0;        // total is float
var name = "Fred";      // name is string
```

  - **Visual Basic, ML, Haskell, and F#** also use type inferencing. The context of the appearance of a variable determines its type

5.4.2.2 Dynamic Type Bindings

- With dynamic type binding, the type of a variable is not specified by a declaration statement, nor can it be determined by the spelling of its name. Instead, the variable is bound to a type when it is assigned a value in an **assignment** statement.
- Dynamic Type Binding: In **Python, Ruby, JavaScript, and PHP**, type binding is dynamic
- Specified through an assignment statement
- Ex, JavaScript

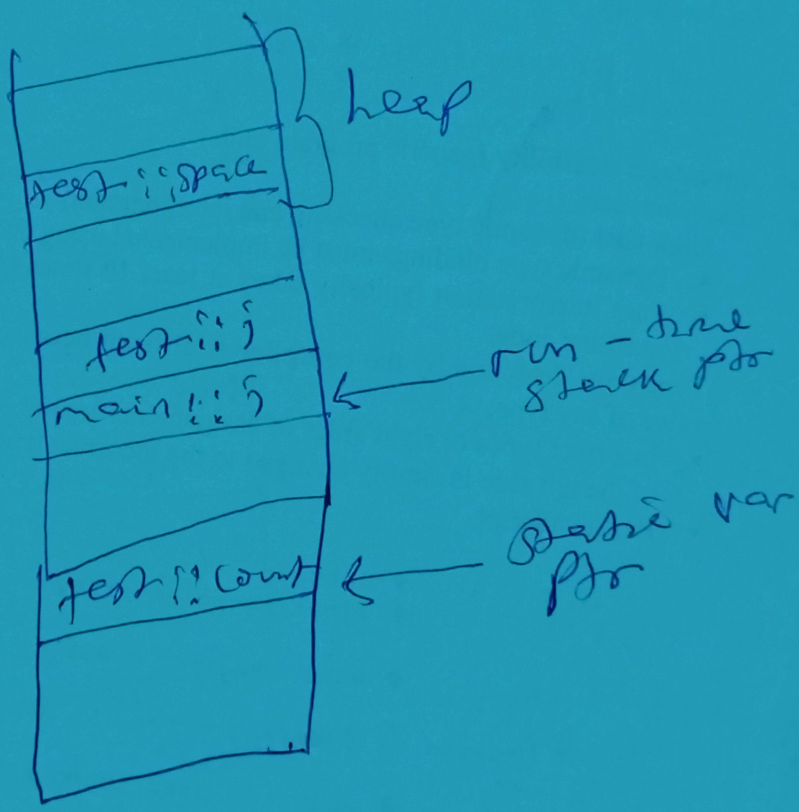
```
list = [2, 4.33, 6, 8];  → single-dimensioned array
list = 47;               → scalar variable
```

- Advantage: **flexibility** (generic program units)
- Disadvantages:
  - **High cost** (dynamic type checking and interpretation)
    - Dynamic type bindings must be implemented using pure interpreter **not** compilers.
    - Pure interpretation typically takes at least **10** times as long as to execute equivalent machine code.
  - Type error detection by the **compiler** is difficult because any variable can be assigned a value of any type.
    - Incorrect types of right sides of assignments are not detected as errors; rather, the type of the left side is simply changed to the incorrect type.
    - Ex, JavaScript

```
i, x        → Integer
y           → floating-point array

i = x;      → what the user meant to type

but because of a keying error, it has the assignment statement

i = y;      → what the user typed instead
```

    - **No error** is detected by the compiler or run-time system. `i` is simply changed to a floating-point array type. Hence, the result is erroneous. In a static type binding language, the compiler would detect the error and the program would not get to execution.

```
#include <stdio.h>
main(){
  int i;
  test();
}
test(){
  int j;
  char *space;
  static int count = 0;
  space = (char *) malloc(100);
}
```



5.4.3 Storage Bindings and Lifetime

- **Allocation** - getting a cell from some pool of available cells.
- **Deallocation** - putting a cell back into the pool.
- The **lifetime** of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.
- Categories of variables by lifetimes:
  - **static**,
  - **stack-dynamic**,
  - **explicit heap-dynamic**, and
  - **implicit heap-dynamic**

5.4.3.1 Static Variables

- Static variables are bound to memory cells **before** execution begins and remains bound to the same memory cell throughout execution
- e.g. all FORTRAN 77 variables, C **static variables** in functions
- Advantages:
  - **Efficiency** (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocation and deallocation of static variables.
  - **History-sensitive**: have vars retain their values between separate executions of the subprogram.
- Disadvantage:
  - Storage **cannot** be shared among variables.
  - Ex: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

5.4.3.2 Stack-dynamic Variables

- Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.
- Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.
- The variable declarations that appear at the beginning of a **Java method** are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.
- Stack-dynamic variables are allocated from the **run-time stack**.
- If scalar, all attributes except address are statically bound.
  - **Local variables** in C subprograms and Java methods.
- Advantages:
  - Allows recursion: each active copy of the recursive subprogram has its own version of the local variables.
  - In the absence of recursion, it conserves storage b/c all subprograms share the same memory space for their locals.

Scope

Scope of a binding (declaration)

```
int x = 20;
void fun(int x)
{
    {float x;
     x = --;}   which x?
}
int main() {
    int y;
    fun(x);   which x?
}
```

"int x;" has a global scope

"float x;" has a local scope.

Aliases

```
int main()
{
    int i = 0;   // int
    int &j = i;  // int reference (reference to int)
    int *p = &i
```

- Disadvantages:
  - Overhead of allocation and deallocation.
  - Subprograms **cannot** be history sensitive.
  - Inefficient references (indirect addressing) is required b/c the place in the stack where a particular var will reside can only be determined during execution.
- In Java, C++, and C#, variables defined in **methods** are by **default** stack-dynamic.

5.4.3.3 Explicit Heap-dynamic Variables

- Nameless memory cells that are allocated and deallocated by explicit directives "run-time instructions", specified by the programmer, which take effect during execution.
- These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.
- The **heap** is a collection of storage cells whose organization is highly disorganized b/c of the unpredictability of its use.
- e.g. Dynamic objects in C++ (via **new** and **delete**)

```
int *intnode;        // create a pointer
. . .
intnode = new int; // allocates the heap-dynamic variable
. . .
delete intnode;      // deallocates the heap-dynamic variable
                     // to which intnode points
```

  - An explicit heap-dynamic variable of int type is created by the new operator.
  - This operator can be referenced through the pointer, intnode.
  - The var is deallocated by the **delete** operator.
- In Java, all data except the primitive scalars are **objects**.
  - Java objects are explicitly heap-dynamic and are accessed through **reference variables**.
  - Java uses **implicit garbage collection**.
- Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.
- Advantage:
  - Provides for dynamic storage management.
- Disadvantage:
  - Inefficient "Cost of allocation and deallocation" and unreliable "difficulty of using pointer and reference variables correctly"

~~Explicit~~

unsigned long j // j is explicitly declared as a 32 bit unsigned integer.

auto j = 65 // j is implicitly declared as a signed 16-bit integer

5.4.3.4 Implicit Heap-dynamic Variables

- Bound to heap storage only when they are assigned value. Allocation and deallocation caused by **assignment** statements.
- All their attributes are bound every time they are assigned.
- e.g. all variables in APL; all strings and arrays in Perl and JavaScript, and PHP.
- Ex, JavaScript

```
highs = [74, 84, 86, 90, 71]; → an array of 5 numeric values
```

- Advantage:
  - Flexibility allowing generic code to be written.
- Disadvantages:
  - Inefficient, because all attributes are dynamic "run-time."
  - Loss of error detection by the compiler.

5.5 Scope 211

- The scope of a variable is the range of statements in which the variable is visible.
- A variable is **visible** in a statement if it can be referenced in that statement.
- **Local variable** is local in a program unit or block if it is declared there.
- **Non-local variable** of a program unit or block are those that are visible within the program unit or block but are not declared there.

5.5.1 Static Scope

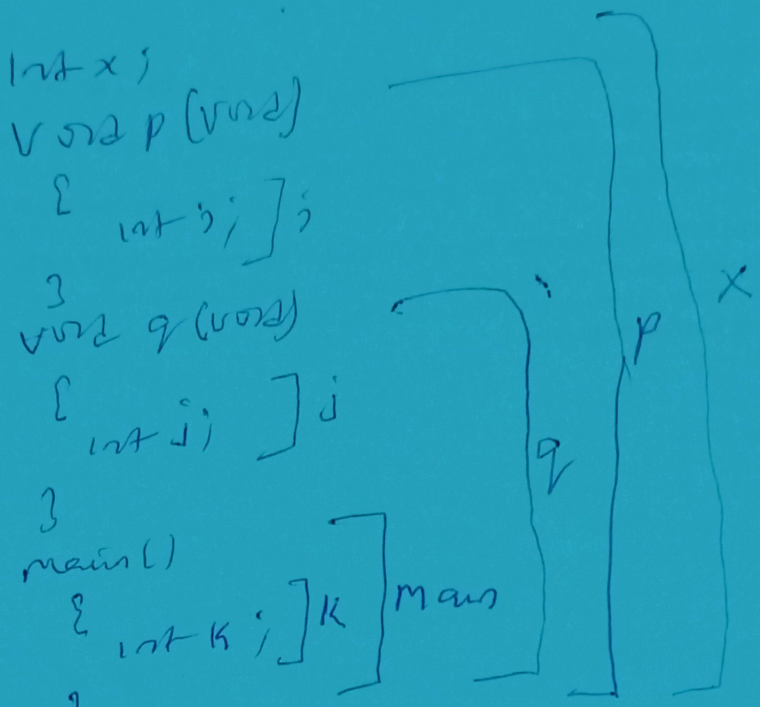
- ALGOL 60 introduced the method of binding names to non-local vars is called **static scoping**.
- Static scoping is named because the scope of a variable can be statically determined – that is **prior** to execution.
- This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
- There are two categories of static scoped languages:
  - Nested Subprograms.
  - Subprograms that cannot be nested.
- Ada, and JavaScript, Common Lisp, Scheme, F#, and Python allow **nested** subprograms, but the C-based languages do **not**.
- When a compiler for static-scoped language finds a reference to a var, the attributes of the var are determined by finding the statement in which it was declared.
- For example:
  - Suppose a reference is made to a var `x` in subprogram `sub1`. The correct declaration is found by first searching the declarations of subprogram `sub1`.
  - If no declaration is found for the var there, the search continues in the declarations of the subprogram that declared subprogram `sub1`, which is called its **static parent**.
    - If a declaration of `x` is not found there, the search continues to the next larger enclosing unit (the unit that declared `sub1`'s parent), and so forth, until a declaration for x is found or the largest unit's declarations have been searched without success.
      ➔ an undeclared var error has been detected.
  - The static parent of subprogram `sub1`, and its static parent, and so forth up to and including the main program, are called the static **ancestors** of `sub1`.
- Ex: JavaScript function, `big`, in which the two functions `sub1` and `sub2` are nested:

```
function big() {
   function sub1() {
      var x = 7;
      sub2();
   }
   function sub2() {
      var y = x;
   }
   var x = 3;
   sub1();
}
```

- Under static scoping, the reference to the variable `x` in `sub2` is to the `x` declared in the procedure `big`.
  - This is true because the search for `x` begins in the procedure in which the reference occurs, `sub2`, but no declaration for `x` is found there.
  - The search thus continues in the static parent of `sub2`, `big`, where the declaration of `x` is found.
  - The `x` declared in `sub1` is ignored, because it is **not** in the static ancestry of `sub2`.
- The variable `x` is declared in both `big` and `sub1`, which is nested inside `big`.
  - Within `sub1`, every simple reference to `x` is to the local `x`.
  - The outer `x` is **hidden** from `sub1`

Example of scope in C

```
int x;
void p(void)
{
    int i;
}
void q(void)
{
    int j;
}
main()
{
    int k;
}
```



MAIN
Declaration of X
SUB1
- declaration of X

Call SUB2

SUB2
---
reference to X

Call SUB1

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses X

Static scoping
Reference to x in SUB2 is to MAIN's X

Dynamic scoping
Reference to X in SUB2 is to SUB1's X

5.5.2 Blocks

- From ALGOL 60, allows a section of code to have its own local variables whose scope is minimized.
- Such variables are **stack dynamic**, so they have their storage allocated when the section is entered and deallocated when the section is exited.
- The **C-based** languages allow any compound statement (a statement sequence surrounded by matched braces) to have declarations and thereby defined a new scope.
- Ex: Skeletal C function:

```
void sub() {
  int count;
  . . .
  while (. . .) {
      int count;
      count ++;
      . . .
  }
  . . .
}
```

- The reference to `count` in the while loop is to that loop's local `count`. The `count` of `sub` is **hidden** from the code inside the while loop.
- A declaration for a var effectively hides any declaration of a variable with the same name in a larger enclosing scope.
- Note that this code is **legal** in C and C++ but **illegal** in Java and C#

- Most functional languages (Scheme, ML, and F#) include some form of **let** construct
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- Ex. Scheme:

```
(LET (
  (name1 expression1)
  . . .
  (namen expressionn)
)
```

  - Consider the following call to LET:

```
(LET (
  (top (+ a b))
  (bottom (- c d))
  (/ top bottom)
)
```

  - This call computes and returns the value of the expression (a + b) / (c – d)

5.5.3 Declaration Order

- C99, C++, Java, and C# allow variable declarations to appear **anywhere** a statement can appear
- In C99, C++, and Java, the scope of all local variables is **from** the declaration to the end of the block
- In C#, the scope of any variable declared in a block is the **whole** block, regardless of the position of the declaration in the block
- However, a variable still must be declared before it can be used
- For example, consider the following C# code:

```
{int x;
  . . .
  {int x;   // Illegal
     . . .
  }
  . . .
}
```

  - Because the scope of a declaration is the whole block, the following nested declaration of x is also illegal:

```
{
  . . .
  {int x;   // Illegal
     . . .
  }
 int x;
}
```

  - Note that C# stall requires that all be declared before they are used

- In C++, Java, and C#, variables can be declared in for statements
  - The scope of such variables is restricted to the for construct

```
void fun() {
  . . .
  for (int count = 0; count < 10; count++) {
     int count;
     . . .
  }
  . . .
}
```

  - The scope of count is from the for statement to the end of for its body (the right brace)

5.5.4 Global Scope

- C, C++, PHP, and Python support a program structure that consists of a sequence of function definitions in a file
  - These languages allow variable declarations to appear outside function definitions
- For example, C and C++ have both declarations and definitions of global data
  - A declaration outside a function definition specifies that it is defined in another file
  - A global variable in C is implicitly visible in all subsequent functions in the file.
  - A global variable that is defined after a function can be made visible in the function by declaring it to be external, as the in the following:

```
extern int sum;
```

5.5.5 Evaluation of Static Scoping

- Works well in many situations
- Problems:
  - In most cases, it allows more access to both variables and subprograms that is necessary
  - As a program evolves, the initial structure is destroyed and local variables often become **global**; subprograms also gravitate toward become global, rather than nested
- An alternative to the use of static scoping to control access to variables and subprograms is an **encapsulation** construct.

5.5.6 Dynamic Scope

- The scope of variables in APL, SNOBOL4, and the early versions of LISP is dynamic. **Perl** and Common Lisp also allow variables to be declared to have dynamic scope, although the default scoping mechanism is these languages is static.
- Dynamic Scoping is based on **calling sequences** of program units, not their textual layout (temporal versus spatial) and thus the scope is determined only at **run time**.
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.
- Ex: Consider again the function `big` from Section 5.5.1, which the two functions `sub1` and `sub2` are nested:

```
function big() {
   function sub1() {
      var x = 7;
      . . .
   }
   function sub2() {
      var y = x;
      . . .
   }
   var x = 3;
   . . .
}
```

- Consider the two different call sequences for `sub2`:
  - `big` calls `sub2 and sub2` use `x`
    - The dynamic parent of `sub2` is `big`. The reference is to the `x` in **big**.
  - `big` calls `sub1`, `sub1` calls `sub2, and sub2` use `x`
    - The search proceeds from the local procedure, `sub2`, to its caller, **sub1**, where a declaration of `x` is found.
  - Note that **if static scoping** was used, in either calling sequence the reference to `x` in `sub2` is to `big's x`.

5.5.7 Evaluation of Static Scoping

- Advantage: convenience
- Disadvantages:
  - While a subprogram is executing, its variables are visible to **all** subprograms it calls
  - Inability to **type check** references to nonlocals statically
  - Difficult to read, because the calling sequence of subprograms must be known to determine the meaning of references to nonlocal variables
  - Finally, accesses to nonlocal variables in dynamic-scoped languages take for **longer** than access to nonlocals when static scoping is used

5.6 Scope and Lifetime 222

- Scope and lifetime are sometimes closely related, but are different concepts
- For example, In a Java method
  - The scope of such a variable is from its **declaration** to the end of the method
  - The lifetime of that variable is the period of **time** beginning when the method is entered and ending when execution of the method terminates
- Consider a **static** variable in a C or C++ function
  - Statically bound to the scope of that function and is also statically bound to storage
  - Its scope is static and local to the function but its lifetime extends over the **entire** execution of the program of which it is a part
- Ex: C++ functions

```
void printheader() {
   . . .
}     /* end of printheader */
void compute() {
   int sum;
   . . .
   printheader();
}     /* end of compute */
```

  - The **scope** of `sum` in contained within `compute` function
  - The **lifetime** of `sum` extends over the time during which `printheader` executes.
  - Whatever storage location `sum` is bound to before the call to `printheader`, that binding will continue during and after the execution of `printheader`.

5.7 Referencing Environments 223

- The referencing environment of a statement is the **collection** of all names that are **visible** in the statement
- In a **static-scoped** language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.
- A subprogram is **active** if its execution has begun but has not yet terminated.
- In a **dynamic-scoped** language, the referencing environment is the local variables plus all visible variables in all active subprograms.
- Ex, Python skeletal, **static-scoped language**

```
g = 3;                      # A global

def sub1():
    a = 5;                  # Crates a local
    b = 6;                  # Crates another local
    . . .                   ← 1
    def sub2():
        global g;           # Global g is now assignable here
        c = 9;              # Creates a new local
        . . .               ← 2
        def sub3():
            nonlocal c;     # Makes nonlocal c visible here
            g = 11;         # Creates a new local
            . . .           ← 3
```

- The referencing environments of the indicated program points are as follows:

Point	Referencing Environment
1	local `a` and `b` (of `sub1`), global `g` for reference, but not for assignment
2	local `c` (of `sub2`), global `g` for both reference and for assignment **Note**: a and b (of sub1) for reference, but not for assignment
3	nonlocal `c` (of `sub2`), local `g` (of `sub3`) **Note**: a and b (of sub1) for reference, but not for assignment

- Ex, **Dynamic-scoped language**
- Consider the following program; assume that the only function calls are the following: `main` calls `sub2`, which calls `sub1`

```
void sub1( ) {
   int a, b;
   . . .                          ← 1
}     /* end of sub1 */
void sub2( ) {
   int b, c;
   . . .                          ← 2
   sub1();
}     /* end of sub2 */
void main( ) {
   int c, d;
   . . .                          ← 3
   sub2( );
}     /* end of main */
```

- The referencing environments of the indicated program points are as follows:

Point	Referencing Environment
1	`a` and `b` of `sub1`, `c` of `sub2`, `d` of `main` (`c` of `main`, `b` of `sub2` hidden)
2	`b` and `c` of `sub2`, `d` of `main` (`c` of `main` is hidden)
3	`c` and `d` of `main`

5.8 Named Constants 224

- It is a variable that is bound to a value only at the time it is bound to storage; its value **cannot** be change by assignment or by an input statement.
- Ex, Java

```
final int LEN = 100;
```

- Advantages: readability and modifiability

Variable Initialization

- The binding of a variable to a value at the time it is bound to storage is called initialization.
- Initialization is often done on the declaration statement.
- Ex, C++

```
int    sum = 0;
int*   ptrSum = &sum;
char   name[] = "George Washington Carver";
```

Summary

- Variables are characterized by the 6 of attributes:
  - Name
  - Address
  - Value
  - Type
  - Lifetime
  - Scope

- Binding is the association of attributes with program entities. Binding can be static or dynamic type binding.
  - **Static type binding**:
    - A binding is **static** if it first occurs **before** run time and remains unchanged throughout program execution.
    - Declaration either explicit or implicit, provide a means of specifying the static binding of variables to types
  - **Dynamic type binding**:
    - A binding is **dynamic** if it first occurs **during** execution or can change during execution of the program.
    - It allows greater flexibility but at the expense of readability, efficiency, and reliability
    -
- Scalar variables can be separated into 4 categories:
  - **Static Variables**
  - **Stack Dynamic Variables**
  - **Explicit Heap Dynamic Variables**
  - **Implicit Heap Dynamic Variables**

- The scope of a variable is the range of statements in which the variable is visible.
  - **Static scope**:
    - Static scoping is named because the scope of a variable can be **statically** determined – that is **prior** to execution
    - This permits a human program reader (and a compiler) to determine the type of every variable in the program simply by examining its source code.
    - It provides a simple, reliable, and efficient method of allowing visibility of nonlocal variables in subprograms
  - **Dynamic scope**:
    - It is based on **calling sequences** of program units, not their textual layout and thus the scope is determined only at **run time**.
    - It provides more flexibility than static scoping but, again, at expense of readability, reliability, and efficiency

Chapter 7

Expressions and Assignment Statements

Chapter 7

Expressions and Assignment Statements

7.1 Introduction 302

- Expressions are the fundamental means of specifying computations in a programming language.
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation.
- Essence of imperative languages is dominant role of assignment statements.

7.2 Arithmetic Expressions 302

- Arithmetic evaluation was one of the motivations for the development of the first programming languages.
- Most of the characteristics of arithmetic expressions in programming languages were inherited from conventions that had evolved in mathematics.
- Arithmetic expressions consist of operators, operands, parentheses, and function calls.
- An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.
  - C-based languages include a **ternary** operator, which has three operands (conditional expression).
- The purpose of an arithmetic expression is to specify an arithmetic computation.
- An implementation of such a computation must cause two actions:
  - Fetching the operands from memory
  - Executing the arithmetic operations on those operands.
- Design issues for arithmetic expressions:
  - What are the operator **precedence** rules?
  - What are the **operator associativity** rules?
  - What is the **order of operand evaluation**?
  - Are there restrictions on operand evaluation **side effects**?
  - Does the language allow user-defined **operator overloading**?
  - What **mode mixing** is allowed in expressions?

7.2.1 Operator Evaluation Order

- Precedence
  - The operator precedence rules for expression evaluation define the order in which the operators of different precedence levels are evaluated.
  - Many languages also include unary versions of addition and subtraction.
  - Unary addition (+) is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand.
  - In Java and C#, unary minus also causes the implicit conversion of `short` and `byte` operands to `int` type.
  - In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is **parenthesized** to prevent it from being next to another operator. For example, unary minus operator (-):

```
A + (- B) * C          // is legal
A + - B * C            // is illegal
```

  - Exponentiation has higher precedence that unary minus, so

```
-A ** B
```

    Is equivalent to

```
-(A ** B)
```

  - The precedences of the arithmetic operators of Ruby and the C-based languages are as follows:

	Ruby	*C-Base Languages*
Highest	`**`	`postfix ++, --`
	`unary +, -`	`prefix ++, --, unary +, -`
	`*, /, %`	`*, /, %`
Lowest	`binary +, -`	`binary +, -`

- Associativity
  - The operator associativity rules for expression evaluation define the order in which adjacent operators with the **same precedence** level are evaluated. An operator can be either left or right associative.
  - Typical associativity rules:
    - Left to right, except **, which is right to left
    - Sometimes unary operators associate right to left (e.g., Fortran)
  - Ex: Java

```
a - b + c              // left to right
```

  - Ex: Fortran

```
A ** B ** C            // right to left

(A ** B) ** C          // in Ada it must be parenthesized
```

  - The associativity rules for a few common languages are given here:

Language	Associativity Rule
Ruby, Fortran	`Left: *, /, +, -`
	`Right: **`
C-based languages	`Left: *, /, %, binary +, binary -`
	`Right: ++, --, unary -, unary +`

  - **APL** is different; **all** operators have equal precedence and all operators associate **right to left**.
  - Ex: APL

```
A x B + C              // A = 3, B = 4, C = 5   ➔   27
```

  - Precedence and associativity rules can be overridden with **parentheses**.

- Parentheses
  - Programmers can alter the precedence and associativity rules by placing parentheses in expressions.
  - A parenthesized part of an expression has precedence over its adjacent unparenthesized parts.
  - Ex:

```
(A + B) * C            // addition will be evaluated first
```

- Expressions in Lisp
  - All arithmetic and logic operations are by explicitly called subprograms
  - Ex: to specify the c expression `a + b * c` in Lisp, one must write the following expression:

```
(+ a (* b c))          // + and * are the names of functions
```

- Conditional Expressions
  - Sometimes **if-then-else** statements are used to perform a conditional expression assignment.
  - Ex: C-based languages

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

  - In the C-based languages, this can be specified more conveniently in an assignment statement using a conditional expression. Note that ? is used in conditional expression as a ternary operator (3 operands).

```
expression_1 ? expression_2 : expression_3
```

  - Ex:

```
average = (count == 0) ? 0 : sum / count;
```

7.2.2 Operand evaluation order

- Operand evaluation order:
  - Variables: fetch the value form memory
  - Constants: sometimes a fetch from memory; sometimes the constant in the machine language instruction and not require a memory fetch.
  - Parenthesized expression: evaluate all operands and operators first
- Side Effects
  - A side effect of a function, called a **functional side effect**, occurs when the function changes either one of its parameters or a global variable.
  - Ex:

```
a + fun(a)
```

    - If `fun` does not have the side effect of changing `a`, then the order of evaluation of the two operands, `a` and `fun(a)`, has no effect on the value of the expression. However, if `fun` **changes** `a`, there is an effect.
  - Ex:

    Consider the following situation: `fun` returns `10` and changes its parameter to have the value `20`, and:

```
a = 10;
b = a + fun(a);
```

    - If the value of `a` is fetched first (in the expression evaluation process), its value is `10` and the value of the expression is **20** (`a + fun(a)` = **10** + 10).
    - But if the second operand is evaluated first, then the value of the first operand is `20` and the value of the expression is **30** (`a + fun(a)` = **20** + 10).
    - The following shows a **C** program which illustrate the same problem.

```
int a = 5;
int fun1() {
   a = 17;
   return 3;
} /* end of fun1 */
void main() {
   a = a + fun1();          // C language a = 20; Java a = 8
} /* end of main */
```

    The value computed for `a` in `main` depends on the order of evaluation of the operands in the expression `a + fun1()`. The value of `a` will be either **8** (if `a` is evaluated first) or **20** (if the function call is evaluated first).

  - Two possible solutions to the functional side effects problem:
    - Write the language definition to disallow functional side effects
    - Write the language definition to demand that operand evaluation order be fixed
  - Java guarantees that operands are evaluated in **left-to-right order**, eliminating this problem.

7.3 Overloaded Operators 311

- The use of an operator for **more than one purpose** is operator overloading.
- Some are common (e.g., + for `int` and `float`).
- Java uses + for addition and for **string catenation**.
- Some are potential trouble (e.g., `&` in C and C++)

```
x = &y    // & as unary operator is the address of y
          // &  as binary operator bitwise logical AND
```

  - Causes the address of `y` to be placed in `x`.
  - Some loss of readability to use the same symbol for two completely unrelated operations.
  - The simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler "difficult to diagnose".

- C++, C#, and F# allow **user-defined** overloaded operators
  - When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)
  - Potential problems:
    - Users can define nonsense operations
    - Readability may suffer, even when the operators make sense

- A **narrowing** conversion is one that converts an object to a type that cannot include all of the values of the original type e.g., `double` to `float`.
- A **widening** conversion is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`.
- Coercion in Expressions
  - A **mixed-mode expression** is one that has operands of different types.
  - A coercion is an **implicit** type conversion.
  - Disadvantage of coercions:
    - They decrease in the type error detection ability of the compiler
  - In most languages, all numeric types are coerced in expressions, using widening conversions
  - In ML and F#, there are **no** coercions in expressions
  - Language designers are not in agreement on the issue of coercions in arithmetic expressions
    - Those against a broad range of coercions are concerned with the reliability problems that can result from such coercions, because they eliminate the benefits of type checking
    - Those who would rather include a wide range of coercions are more concerned with the loss in flexibility that results from restrictions.
    - The issue is whether programmers should be concerned with this category of errors or whether the compiler should detect them.
  - Ex: Java

```
int a;
float b, c, d;
. . .
d = b * a;
```

    - Assume that the second operand of the multiplication operator was supposed to be `c`, but because of a keying error it wat typed as `a`
    - Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. Simply, `b` will be coerced to `float`.
- Explicit Type Conversions
  - In the C-based languages, explicit type conversions are called **casts**
  - Ex: In Java, to specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

```
(int)angle
```

- Errors in Expressions
  - Caused by:
    - Inherent limitations of arithmetic — e.g. division by zero
    - Limitations of computer arithmetic — e.g. overflow or underflow
  - Floating-point overflow and underflow, and division by zero are examples of **run-time errors**, which are sometimes called exceptions.

7.5 Relational and Boolean Expressions 316

Relational Expressions

- A relational operator: an operator that compares the values of its two operands
- Relational Expressions: two operands and one relational operator
- The value of a relational expression is Boolean, unless it is not a type included in the language
  - Use relational operators and operands of various types
  - Operator symbols used vary somewhat among languages (`!=`, `/=`, `.NE.`, `<>`, `#`)
- The syntax of the relational operators available in some common languages is as follows:

Operation	*Ada*	*C-Based Languages*	*Fortran 95*
Equal	=	==	.EQ. or ==
Not Equal	/=	!=	.NE. or <>
Greater than	>	>	.GT. or >
Less than	<	<	.LT. or <
Greater than or equal	>=	>=	.GE. or >=
Less than or equal	<=	<=	.LE. or >=

- JavaScript and PHP have two additional relational operator, `===` and `!==`
  - Similar to their cousins, `==` and `!=`, except that they do not coerce their operands

```
"7" == 7      // true in JavaScript
"7" === 7     // false in JavaScript, because no coercion is done on the operand
                 of this operator
```

Boolean Expressions

- Operands are Boolean and the result is **Boolean**

FORTRAN 77	*FORTRAN 90*	*C*	*Ada*
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not

- Versions of C prior to C99 have **no** Boolean type; it uses `int` type with 0 for false and **nonzero** for true.
- One odd characteristic of C's expressions: `a > b > c` is a legal expression, but the result is not what you might expect

```
a > b > c
```

  - The left most operator is evaluated first because the relational operators of C are **left** associative, producing either 0 **or** 1
  - Then, this result is compared with var `c`. There is **never** a comparison between `b` and `c`.

7.6 Short-Circuit Evaluation 318

- A short-circuit evaluation of an expression is one in which the result is determined **without** evaluating all of the operands and/or operators.
- Ex:

```
(13 * a) * (b/13 - 1)        // is independent of the value of (b/13 - 1)
                             if a = 0, because 0 * x = 0 for any x
```

  - So when `a = 0`, there is no need to evaluate `(b/13 - 1)` or perform the second multiplication.
  - However, this shortcut is not easily detected during execution, so it is never taken.
- The value of the Boolean expression:

```
(a >= 0) && (b < 10)         // is independent of the second expression
                             if a < 0, because expression (FALSE && (b < 10))
                             is FALSE for all values of b
```

  - So when a < 0, there is **no** need to evaluate b, the constant 10, the second relational expression, or the && operation.
  - Unlike the case of arithmetic expressions, this shortcut can be easily discovered during execution.
- Short-circuit evaluation exposes the potential problem of side effects in expressions

```
(a > b) || (b++ / 3)         // b is changed only when a <= b
```

- If the programmer assumed `b` would change every time this expression is evaluated during execution, the program will fail.
- C, C++, and Java: use short-circuit evaluation for the usual Boolean operators (**&&** and **||**), but also provide **bitwise** Boolean operators that are **not** short circuit (**&** and **|**)

Simple Assignments

- The C-based languages use == as the equality relational operator to avoid confusion with their assignment operator
- The operator symbol for assignment:
  1. = Fortran, Basic, PL/I, C, C++, Java
  2. := ALGOL, Pascal, Ada

Conditional Targets

- Ex: Perl

```
($flag ? $count1 : $count2) = 0;  ⇔   if ($flag) {
                                          $count1 = 0;
                                      } else {
                                          $count2 = 0;
                                      }
```

Compound Assignment Operators

- A compound assignment operator is a shorthand method of specifying a commonly needed form of assignment
- The form of assignment that can be abbreviated with this technique has the destination variable also appearing as the first operand in the expression on the right side, as in

```
a = a + b
```

- The syntax of assignment operators that is the catenation of the desired binary operator to the = operator

```
sum += value;    ⇔    sum = sum + value;
```

Unary Assignment Operators

- C-based languages include two special unary operators that are actually abbreviated assignments
- They combine increment and decrement operations with assignments
- The operators ++ and -- can be used either in expression or to form stand-alone single-operator assignment statements. They can appear as **prefix** operators:

```
sum = ++ count;    ⇔    count = count + 1; sum = count;
```

- If the same operator is used as a **postfix** operator:

```
sum = count ++;    ⇔    sum = count; count = count + 1;
```

Assignment as an Expression

- This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand.
- Ex:

```
while ((ch = getchar())!= EOF) {. . .}
                                    // why ( ) around assignment?
```

  - The assignment statement must be parenthesized because the precedence of the assignment operator is **lower** than that of the relational operators.
- Disadvantage: another kind of expression side effect which leads to expressions that are **difficult** to read and understand. For example

```
a = b + (c = d / b) - 1
```

  denotes the instructions

```
Assign d / b to c
Assign b + c to temp
Assign temp - 1 to a
```

- There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors.

```
if (x = y) . . .
```

  instead of

```
if (x == y) . . .
```

Multiple Assignments

- Perl, Ruby, and Lua provide **multiple-target** multiple-source assignments
- Ex: Perl

```
($first, $second, $third) = (20, 30, 40);
```

  - The semantics is that `20` is assigned to `$first`, `40` is assigned to `$second`, and `60` is assigned to `$third`.

    Also, the following is legal and performs an **interchange**:

```
($first, $second) = ($second, $first);
```

  - The correctly interchanges the values of `$first` and `$second`, `60` **without** the use of a temporary variable

Assignment in Functional Programming Languages

- Identifiers in functional languages are only names of values
- Ex: in ML, names are bound to values with the **`val`** declaration, whose form is exemplified in the following:

```
val cost = quantity * price;
```

  - If `cost` appears on the left side of a subsequent **`val`** declaration, that declaration creates a **new** version of the name `cost`, which has **no** relationship with the previous version, which is then hidden
- F#'s **`let`** is like ML's **`val`**, except **`let`** also creates a new scope

1)

If we write a code in C as

```
int a, b;
float c;
- - - - -
c = a/b;
```

In the above codes, explain how coercions are done during compilation stage.

7.8 Mixed-Mode Assignment 324

- Assignment statements can also be mixed-mode
- In Fortran, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done.
- In Java and C#, only **widening** assignment coercions are done.
- In Ada, there is **no** assignment coercion.
- In all languages that allow mixed-mode assignment, the coercion takes place only **after** the right side expression has been evaluated. For example, consider the following code:

```
int a, b;
float c;
. . .
c = a / b;
```

  - Because `c` is `float`, the values of `a` and `b` could be coerced to `float` before the division, which could produce a different value for `c` than if the coercion were delayed (for example, if `a` were 2 and `b` were 3).

Summary 324

- **Expressions** consist of constants, variables, parentheses, function calls, and operators
- **Assignment** statements include target variables, assignments operators, and expressions
- The **associativity** and **precedence** rules for operators in the expressions of a language determine the order of operator evaluation in those expressions
- Operand evaluation order is important if **functional side effects** are possible
- Type conversions can be widening or narrowing
  - Some narrowing conversions produce erroneous values
  - Implicit type conversions, or coercions, in expressions are common, although they eliminate the error-detection benefit of type checking, thus lowing reliability

Chapter 8

Statement-Level Control Structures

Chapter 8

Statement-Level Control Structures

8.1 Introduction 330

- A control structure is a control statement and the statements whose execution it controls
  - Selection Statements
  - Iterative Statements
- There is only one design issue that is relevant to all of the selection and iteration control statements:
  - Should a control structure have multiple entries?

8.2 Selection Statements 332

- A selection statement provides the means of choosing between two or more paths of execution.
- Selection statement fall into two general categories:
  - Two-way selection
  - Multiple-way selection

8.2.1 Two-Way Selection Statements

- The general form of a two-way selector is as follows:

```
if control_expression
   then clause
   else clause
```

- Design issues
  - What is the form and type of the control expression?
  - How are the `then` and `else` clauses specified?
  - How should the meaning of nested selectors be specified?

- The control expression
  - Control expressions are specified in parenthesis if the `then` reserved word is not used to introduce the `then` clause, as in the C-based languages
  - In C89, which did not have a Boolean data type, **arithmetic** expressions were used as control expressions
  - In contemporary languages, such as Java and C#, **only Boolean** expressions can be used for control expressions

- Clause Form
  - In most contemporary languages, the `then` and `else` clauses either appear as single statements or compound statements.
  - C-based languages use **braces** to form compound statements.
  - One exception is **Perl**, in which all then and else clauses must be **compound** statements, even if they contain single statements
  - In Python and Ruby, clauses are statement sequences
  - Python uses **indentation** to define clauses

```
if x > y :
   x = y
   print " x was greater than y"
```

    - All statements equally indented are included in the compound statement. Notice that rather than `then`, a colon is used to introduce the `then` clause in the Python

- Nesting Selectors
  - In Java and contemporary languages, the static semantics of the language specify that the `else` clause is always paired with the **nearest** unpaired `then` clause

```
if (sum == 0)
    if (count == 0)
        result = 0;
else
    result = 1;
```

    - A rule, rather than a syntactic entity, is used to provide the disambiguation
    - So, in the example, the `else` clause would be the alternative to the second `then` clause
    - To force the alternative semantics in Java, a different syntactic form is required, in which the inner `if` is put in a compound, as in

```
if (sum == 0) {
    if (count == 0)
        result = 0;
}
else
    result = 1;
```

  - C, C++, and C# have the same problem as Java with selection statement nesting
  - Ruby, statement sequences as clauses:

```
if sum == 0 then
    if count == 0 then
        result = 0
    else
        result = 1
    end
end
```

  - Python, all statements uses indentation to define clauses

```
if sum == 0 :
    if count == 0 :
        result = 0
    else :
        result = 1
```

8.2.2 Multiple Selection C

- The multiple selection
statement groups.
- Design Issues
What is the
How are th
Is execu
segme
How
Wh
C

(2)

9 Multiple selection constructs allows the selection of one of any number of statements or statement groups.

Write and discuss the general switch-case statements in C, C#, Java and Javascript.

(b) How does switch statement in C# differ from C-based language?

8.2.2 Multiple Selection Constructs

- The multiple selection construct allows the selection of one of any number of statements or statement groups.

- Design Issues
  - What is the form and type of the control expression?
  - How are the selectable segments specified?
  - Is execution flow through the structure restricted to include just a single selectable segment?
  - How are case values specified?
  - What is done about unrepresented expression values?

- C, C++, Java, and JavaScript switch

```
switch (expression) {
   case constant_expression1 : statement1;
        ...
   case constant_expressionn : statementn;
   [default: statementn+1]
}
```

  - The control expression and the constant expressions some discrete type including integer types as well as character and enumeration types
  - The selectable statements can be statement sequences, blocks, or compound statements
  - Any number of segments can be executed in one execution of the construct (there is no implicit branch at the end of selectable segments)
  - `default` clause is for unrepresented values (if there is no `default`, the whole statement does nothing)
  - Any number of segments can be executed in one execution of the construct (a trade-off between reliability and flexibility—convenience.)
  - To avoid it, the programmer must supply a break statement for each segment.
- C# switch
  - C# switch statement differs from C-based in that C# has static semantic rule disallows the implicit execution of more than one segment
  - The rule is that every selectable segment must end with an explicit unconditional branch statement either a `break`, which transfers control out of the switch construct, or a `goto`, which can transfer control to on of the selectable segments. C# switch statement example:

```
switch (value)   {
   case -1:    Negatives++;
               break;
   case  0:    Positives++;
               goto case 1;
   case  1:    Positives ++;
   default:    Console.WriteLine("Error in switch \n");
}
```

- Multiple Selection Using `if`
  - Multiple Selectors can appear as direct extensions to two-way selectors, using else-if clauses
  - Ex, Python selector statement (note that else-if is spelled `elif` in Python):

```
if count < 10 :
   bag1 = True
elif count < 100 :
   bag2 = True
elif count < 1000 :
   bag3 = True
```

which is equivalent to the following:

```
if count < 10 :
   bag1 = True
else :
   if Count < 100 :
      bag2 = True
else :
     if Count < 1000 :
        bag3 = True
```

    - The `elsif` version is the more readable of the two.

  - The Python example can be written as a Ruby `case`

```
case
   when count < 10        then bag1 = true
   when count < 100       then bag2 = true
   when count < 1000      then bag3 = true
end
```

  - Notice that this example is not easily simulated with a **switch-case** statement, because each selectable statement is chosen on the basis of a Boolean expression
  - In fact, none of the multiple selectors in contemporary languages are as **general** as the if-then-else-if statement

8.3 Iterative Statements 343

- An iterative statement is one that cause a statement or collection of statements to be executed zero, one, or more times
- The repeated execution of a statement or compound statement is accomplished either by iteration or recursion
- An iterative statement is often called **loop**
- Iteration is the very essence of the power of computer
- The repeated execution of a statement is often accomplished in a functional language by recursion rather than by iteration
- General design issues for iteration control statements:
  - How is iteration controlled?
  - Where should the control mechanism appear in the loop statement?
- The primary possibilities for iteration control are logical, counting, or a combination of the two
- The main choices for the location of the control mechanism are the top of the loop or the bottom of the loop
- The **body** of a loop is the collection of statements whose execution is controlled by the iteration statement
- The term **pretest** means that the loop completion occurs before the loop body is executed
- The term **posttest** means that the loop completion occurs after the loop body is executed
- The iteration statement and the associated loop body together form an **iteration statement**

8.3.1 Counter-Controlled Loops

- A counting iterative control statement has a variable, called the **loop variable**, in which the count value is maintained
- It also includes means of specifying the **intial** and **terminal** values of the loop variable, and the difference between sequential loop variable values, called the **stepsize**.
- The intial, terminal and stepsize are called the **loop parameters**.
- Design issues:
  - What are the type and scope of the loop variable?
  - Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control?
  - Should the loop parameters be evaluated only once, or once for every iteration?
  -
- Fortran 90's **DO** syntax:

```
[name:] DO label variable = initial, terminal [, stepsize]
        . . .
END DO [name]
```

  - The label is that of the last statement in the loop body, and the stepsize, when absent, defaults to **1**.
  - Loop variable **must** be an INTEGER and may be either negative or positive.
  - The loop parameters are allowed to be expressions and can have negative or positive values.
  - They are evaluated at the beginning of the execution of the DO statement, and the value is used to compute an iteration count, which then has the number of times the loop is to be executed.
  - The loop is controlled by the iteration count, not the loop param, so even if the params are changed in the loop, which is legal, those changes cannot affect loop control.
  - The iteration count is an internal var that is inaccessible to the user code.
  - The DO statement is a single-entry structure

3

(a) Consider the following statement:

```
for (count1 = 0, count2 = 10;
     count1 < 10 && count2 <= 1000;
     sum = ++count1 + count2, count2 = 2*);
```

What is the operational semantics description of the above code?

(b) What makes the for statement in Java & C# differs from C++?

- The for statement of the C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3])
   loop body
```

  - The loop body can be a single statement, a compound statement, or a null statement

```
for (count = 1; count <= 10; count++)
   . . .
```

  - All of the expressions of C's `for` are optional
  - If the second expression is absent, it is an **infinite** loop
  - If the first and third expressions are absent, no assumptions are made
  - The C-based languages for design choices are:
    - There are no explicit loop variable or loop parameters
    - All involved variables can be changed in the loop body
    - First expression is evaluated once, but the other two are evaluated with each iteration
    - It is legal to branch into the body of a for loop in C
  - C's for is more **flexible** than the counting loop statements of Fortran and Ada, because each of the expressions can comprise multiple statements, which in turn allow multiple loop variables that can be of any type
  - Consider the following for statement:

```
for (count1 = 0, count2 = 1.0;
     count1 <= 10 && count2 <= 100.0;
     sum = ++count1 + count2, count2 *= 2.5)
   ;
```

The operational semantics description of this is:

```
count1 = 0
count2 = 1.0
loop:
   if count1 > 10 goto out
   if count2 > 100.0 goto out
   count1 = count1 + 1
   sum = count1 + count2
   count2 = count2 * 2.5
   goto loop
out:
   . . .
```

  - The loop above does not need and thus does **not** have a loop body
  - The for statement of C99 and C++ differs from earlier version of C in two ways:
    - It can use an arithmetic expression or a Boolean expression for loop control
    - The first expression can include variable definitions (scope is from the definition to the end of the loop body), for example

```
for (int count = 0; count <= 10; count++) { . . . }
```

  - The for statement of Java and C# is like that of C++, except that the loop control expression is restricted to Boolean

- The for statement of Python
  - The general form of Python's `for` is:

```
for loop_variable in object:
    - loop body
[else:
    - else clause]
```

  - The `object` is often range, which is either a list of values in brackets (`[2, 4, 6]`), or a call to the range function (`range(5)`, which returns `0, 1, 2, 3, 4`)
  - The loop variable takes on the values specified in the given range, one for each iteration
  - The else clause, which is optional, is executed if the loop terminates normally
  - Consider the following example:

```
for count in [2, 4, 6] :
    print count
```

produces

```
2
4
6
```

8.3.2 Logically Controlled Loops

- Repetition control is based on a Boolean expression rather than a counter
- Design Issues:
  - Should the control be pretest or posttest?
  - Should the logically controlled loop be a special form of a counting loop or a separate statement?
- The C-based programming languages include both pretest and posttest logically controlled loops that are not special forms of their counter-controlled iterative statements
- The **pretest** and **posttest** logical loops have the following forms (`while` and `do-while`):

```
while (control_expression)
   loop body
```

and

```
do
   loop body
while (control_expression);
```

- These two statements forms are exemplified by the following C# code:

```
sum = 0;
indat = Int32.Parse(Console.ReadLine( ));
while (indat >= 0) {
   sum += indat;
   indat = Int32.Parse(Console.ReadLine( ));
}

value = Int32.Parse(Console.ReadLine( ));
do {
   value /= 10;
   digits ++;
} while (value > 0);
```

- The only real difference between the do and the while is that the do always causes the loop body to be executed **at least once**
- Java's `while` and `do` statements are similar to those of C and C++, except the control expression must be Boolean type, and because Java does **not** have a goto, the loop bodies cannot be entered anywhere but at their beginning

8.3.3 User-Located Loop Control Mechanisms

- It is sometimes convenient for a programmer to choose a location for loop control other than the top or bottom of the loop
- Design issues:
  - Should the conditional mechanism be an integral part of the exit?
  - Should only one control body be exited, or can enclosing loops also be exited?
- C and C++ have unconditional unlabeled exits (`break`)
- Java, Perl, and C# have unconditional labeled **exits** (`break` in Java and C#, `last` in Perl)
- The following is an example of nested loops in C#:

```
OuterLoop:
   for (row = 0; row < numRows; row++)
      for (col = 0; col < numCols; col++) {
         sum += mat[row][col];
         if (sum > 1000.0)
               break outerLoop;
   }
```

- C and C++ include an unlabeled control statement, `continue`, that transfers control to the control mechanism of the smallest enclosing loop
- This is not an exit but rather a way to **skip** the rest of the loop statements on the current iteration without terminating the loop structure. Ex:

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) continue;
   sum += value;
}
```

- A negative value causes the assignment statement to be **skipped**, and control is transferred instead to the conditional at the top of the loop
- On the other hand, in

```
while (sum < 1000) {
   getnext(value);
   if (value < 0) break;
   sum += value;
}
```

- A negative value **terminates** the loop

- Java, Perl, and C# have statements similar to continue, except they can include labels that specify which loop is to be continued
- The motivation for user-located loop exits is simple: They fulfill a common need for goto statements through a highly restricted branch statement
- The target of a goto can be many places in the program, both above and below the goto itself
- However, the targets of user-located loop exits must be below the exit and can only follow immediately the end of a compound statement

8.3.4 Iteration Based on Data Structures

- A general data-based iteration statement uses a user-defined data structure and a user-defined function (the **iterator**) to go through the structure's elements
- The iterator is called at the beginning of each iteration, and each time it is called, the iterator return a n element from a particular data structure in some specific order
- C's `for` can be used to build a user-defined iterator:

```
for (ptr=root; ptr==NULL; ptr = traverse(ptr)) {
    . . .
}
```

- Java 5.0 uses `for`, although it is called `foreach`
  - The following statement would iterate though all of its elements, setting each to myElement:

```
for (String myElement : myList) { . . . }
```

  - The new statement is referred to as "`foreach`," although is reserved word is `for`

- C#'s `foreach` statement iterates on the elements of array and other collections
  - C# and F# (and the other .NET languages) have generic library classes, like Java 5.0 (for arrays, lists, stacks, and queues)
  - For example, there are **generic** collection classes for lists, which are dynamic length array, stacks, queues, and dictionaries (has table)
  - All of these predefined generic collections have built-in iterator that are used implicitly with the `foreach` statement
  - Furthermore, users can define their own collections and write their own iterators, implement the `IEnumerator` interface, which enables the use of use `foreach` on these collections

```
List<String> names = new List<String>();
names.Add("Bob");
names.Add("Carol");
names.Add("Ted");
foreach (Strings name in names)
        Console.WriteLine (name);
```

8.4 Unconditional Branching 355

- An unconditional branch statement transfers execution control to a specified place in the program
- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements
- However, using the goto carelessly can lead to **serious problems**
- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly **unreliable** and costly to **maintain**
- There problems follow directly from a goto's ability to force any program statement to follow any other in execution sequence, regardless of whether the statement proceeds or follows previously executed statement in textual order
- Java, Python, and Ruby do **not** have a goto. However, most currently popular languages include a goto statement
- C# uses goto in the **switch** statement

8.5 Guarded Commands 356

- New and quite different forms of selection and loop structures were suggested by Dijkstra (1975)
- His primary motivation was to provide control statements that would support a **new** program design methodology that ensured correctness (verification) during development rather than when verifying or testing completed programs
- Basis for two linguistic mechanisms for concurrent programming in CSP (Hoare, 1978)
- Basic idea: if the order of evaluation is not important, the program should **not** specify one
- Dijkstra's selection guarded command has the form

```
if <Boolean expr> -> <statement>
[] <Boolean expr> -> <statement>
 ...
[] <Boolean expr> -> <statement>
fi
```

- Semantics: when construct is reached,
  - Evaluate all Boolean expressions
  - If more than one are true, choose one **non-deterministically**
  - If none are true, it is a runtime error
- Ex

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

  - If `i = 0` and `j > i`, this statement chooses non-deterministically between the first and third assignment statements
  - If `i` is equal to `j` and is not zero, a run-time error occurs because none of the condition are true

- Ex

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

  - This computes the desired result without over specifying the solution
  - In particular, if x and y are equal, it does **not** matter which we assign to max
  - This is a form of abstraction provide by the non-deterministic semantics of the statement

- *The loop structure proposed by Dijkstra has the form*

```
do <Boolean> -> <statement>
[] <Boolean> -> <statement>
 ...
[] <Boolean> -> <statement>
od
```

- Semantics: for each iteration
  - Evaluate all Boolean expressions
  - If more than one are true, choose one non-deterministically; then start loop again
  - If none are true, exit loop

- Ex Consider the following problem: Given four integer variables, q1, q2, q3, and q4, rearrange the values of the four so that $q1 \leq q2 \leq q3 \leq q$
  - **Without** guarded commands, one straightforward solution is to put the four values into an array, sort the array, and then assign the values from the array back into the scalar variables q1, q2, q3, and q4. While this solution is not difficult, it requires a good deal of code, especially if the sort process must be included.
  - Now, uses guarded commands to solve the same problem but in a more concise and **elegant** way

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

- Dijkstra's guarded command control statements are **interesting**, in part because they illustrate how the syntax and semantics of statements can have an impact on program verification and vice versa.
- Program verification is impossible when **goto** statements are used
- Verification is greatly simplified if
  - only selection and logical pretest loops
  - only guarded commands

Summary 359

- Control statements occur in several categories:
  - Selection Statements
  - Iterative Statements
  - Unconditional branching
- The **switch** statement of the C-based languages is representative of multiple-selection statements
- C's **for** statement is the most flexible iteration statement although its flexibility lead to some reliability problem
- Data-based iterators are loop statements for processing data structures, such as linked lists, hashes, and trees.
  - The **for** statement of the C-based languages allows the user to create iterators for user-defined data
  - The **foreach** statement of Perl and C# is a predefined iterator for standard data structure
- The unconditional branch, or **goto**, is the most powerful statement for controlling the flow of execution of a program's statements
  - The unconditional branch, or goto, has been part of **most** imperative languages
  - Its problems have been widely discussed and debated.
  - The current consensus is that it should remain in most languages but that its **dangers** should be **minimized** through programming discipline
- Dijkstra's **guarded commands** are alternative control statement with positive theoretical characteristics.
  - Although they have not been adopted as the control statements of a language, part of the semantics appear in the concurrency mechanisms of CSP and the function definitions of Haskell

Chapter 9

Subprograms

Chapter 9

Subprograms

9.1 Introduction 366

- Subprograms are the fundamental building blocks of programs and are therefore among the most import concepts in programming language design
- This **reuse** results in savings, including memory space and coding time

9.2 Fundamentals of Subprograms 366

9.2.1 General Subprogram Characteristics

- Each subprogram has a single entry point
- The caller is suspended during execution of the called subprogram, which implies that there is **only one** subprogram in execution at any given time
- Control always returns to the caller when the called subprogram's execution terminates

9.2.2 Basic Definitions

- A subprogram definition describes the interface to and the actions of the subprogram abstraction
- A subprogram call is an explicit request that the called subprogram be executed
- A subprogram is said to be **active** if, after having been called, it has begun execution but has not yet completed that execution
- The **two** fundamental types of the subprograms are:
  - Procedures
  - Functions
- A subprogram **header** is the first line of the definition, serves several purposes:
  - It specifies that the following syntactic unit is a subprogram definition of some particular kind
  - If the subprogram is not anonymous, the header provides a name for the subprogram
  - It may optionally specify a list of parameters.
- Consider the following header examples:
  - In Python, the header of a subprogram named `adder`

```
def adder(parameters):
```

  - Ruby subprogram headers also being with `def`
  - The header of a JavaScript subprogram begins with `function`

- Ada

```
procedure adder(parameters)
```

- In C, the header of a function named `adder` might be as follows:

```
void adder(parameters)
```

- One characteristic of **Python** functions that sets them apart from the functions of other common programming languages is that function `def` statements are **executable**
  - Consider the following skeletal example:

```
if . . .
   def fun(. . . ):
   . . .
else
   def fun(. . . ):
   . . .
```

  - When a `def` statement is executed, it assigns the given name to the given function body
  - If the then clause of this selection construct is executed, that version of the function `fun` can be called, but not the version in the else clause. Likewise, if the else clause is chosen, its version of the function can be called but the one in the then clause cannot
- The parameter profile (sometimes called the **signature**) of a subprogram is the number, order, and types of its formal parameters
- The protocol of a subprogram is its parameter profile plus, if it is a function, its return type
- A subprogram declaration provides the protocol, but not the body, of the subprogram
- Function declarations are common in C and C++ programs, where they are called **prototypes**
- Java and C# do not need declarations of their methods, because there is no requirement that methods be defined before they are called in those languages

9.2.3 Parameters

- Subprograms typically describe computations. There are two ways that a non-local method program can gain access to the data that it is to process:
  - Through **direct access** to non-local variables
    - Declared elsewhere but **visible** in the subprogram
  - Through **parameter passing** "more flexible"
    - Data passed through parameters are accessed through names that are **local** to the subprogram
    - A subprogram with parameter access to the data it is to process is a parameterized computation
    - It can perform its computation on whatever data it receives through its parameters
- A **formal** parameter is a dummy variable listed in the subprogram header and used in the subprogram.
- Subprograms call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram
- An **actual** parameter represents a value or address used in the subprogram call statement

- Actual/formal parameter correspondence:
  - **Positional**: The first actual parameter is bound to the first formal parameter and so forth
  - **Keyword**: The name of the formal parameter is to be bound with the actual parameter.
    - They can appear in any order in the actual parameter list. Python functions can be called using this technique, as in

```
sumer(length = my_length, list = my_array, sum = my_sum)
```

    - Where the definition of `sumer` has the formal parameters `length`, `list`, and `sum`
    - Advantage: parameter order is **irrelevant**
    - Disadvantage: user of the subprogram must know the **names** of formal parameters
- In Python, Ruby, C++, and PHP, formal parameters can have **default** values (if no actual parameter is passed)
  - In C++, which has no keyword parameters, the rules for default parameters are necessarily different
    - The default parameters must appear last; parameters are positionally associated
    - Once a default parameter is omitted in a call, all remaining formal parameters must have default values

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

    - An example call to the C++ `compute_pay` function is:

```
pay = compute_pay(20000.0, 0.15);
```

- In most languages that do **not** have default values for formal parameters, the number of actual parameters in a call must match the number of formal parameters in the subprogram definition header

9.2.4 Procedures and Functions

- There are **two** distinct categories of subprograms, procedures and functions
- Subprograms are collections of statements that define parameterized computations. Functions return values and procedures do **not**
- **Procedures** can produce results in the calling program unit by two methods:
  - If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure **can** change them
  - If the subprogram has formal parameters that allow the transfer of data to the caller, those parameters **can** be changed
- **Functions:** Functions structurally resemble procedures but are semantically modeled on mathematical functions
  - If a function is a faithful model, it produces **no** side effects
  - It modifies neither its parameters **nor** any variables defined outside the function
  - The **returned** value is its only effect
- The functions in **most** programming languages have side effects
- The **methods** of Java are syntactically similar to the **functions** of C

9.3 Design Issues for Subprograms 374

- Design Issues for Subprograms
  - Are local variables static or dynamic?
  - Can subprogram definitions appear in other subprogram definitions?
  - What parameter passing methods are provided?
  - Are parameter types checked?
  - If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
  - Are functional side effects allowed?
  - What types of values can be returned from functions?
  - How many values can be returned from functions?
  - Can subprograms be overloaded?
  - Can subprogram be generic?
  - If the language allows nested subprograms, are closures supported?

9.4 Local Referencing Environments 375

- Variables that are defined inside subprograms are called **local** variables.
- Local variables can be either static or stack dynamic "bound to storage when the program begins execution and are unbound when execution terminates"
- Local variables can be **stack dynamic**
  - Advantages
    - Support for recursion
    - Storage for locals is shared among some subprograms
  - Disadvantages:
    - Allocation/deallocation time
    - Indirect addressing "only determined during execution"
    - Subprograms cannot be history sensitive "can't retain data values between calls"
- Local variables can be **static**
  - Advantages
    - Static local variables can be accessed faster because there is no indirection
    - No run-time overhead for allocation and deallocation
    - Allow subprograms to be history sensitive
  - Disadvantages
    - Inability to support recursion
    - Their storage can't be shared with the local variables of other inactive subprograms
- In C functions, locals are stack-dynamic unless specifically declared to be static. Ex:

```
int adder(int list[ ], int listlen) {
   static int sum = 0;          //sum is static variable
   int count;                   //count is stack-dynamic variable
   for (count = 0; count < listlen; count++)
   sum += list[count];
   return sum;
}
```

- The methods of C++, Java, and C# have **only** stack-dynamic local variables

9.5.1 Semantic Models of Parameter Passing

- Formal parameters are characterized by one of three distinct semantic models:
  - **in** mode: They can receive data from corresponding actual parameters
  - **out** mode: They can transmit data to the actual parameter
  - **inout** mode: They can do both
- There are two conceptual models of how data transfers take places in parameter transmission:
  - Either an actual **value** is copied (to the caller, to the callee, or both ways), or
  - An access **path** is transmitted
- Most commonly, the access path is a simple **pointer** or **reference**
- Figure below illustrates the three semantics of parameter passing when values are copied

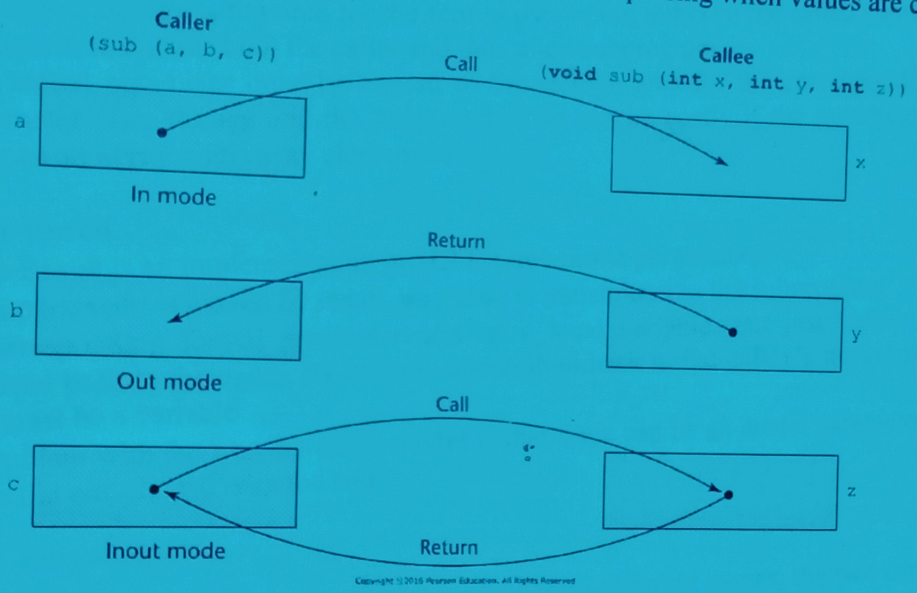


Figure 9.1 The three semantics models of parameter passing when physical moves are used

9.5.2 Implementation Models of Parameter Passing

- Five implementation models of parameter passing:
  - Pass-by-value
  - Pass-by-result
  - Pass-by-value-result
  - Pass-by-reference
  - Pass-by-name

1. Pass-by-Value

- When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, which then acts as a local var in the subprogram, thus implementing **in-mode** semantics
- Disadvantages:
  - Additional storage is required for the formal parameter, either in the called subprogram or in some area outside both the caller and the called subprogram
  - The actual parameter must be **copied** to the storage area for the corresponding formal parameter. The storage and the copy operations can be costly if the parameter is large, such as an **array** with many elements

2. Pass-by-Result

- Pass-by-Result is an implementation model for **out-mode** parameters
- When a parameter is passed by result, **no value** is transmitted to the subprogram
- The corresponding formal parameter acts as a local variable, but just before control is transferred back to the caller, its value is transmitted back to the **caller's actual parameter**, which must be a variable
- One problem with the pass-by-result model is that there can be an actual parameter collision, such as the one created with the call

```
sub(p1, p1)
```

  - In `sub`, assuming the two formal parameters have different names, the two can obviously be assigned different values
  - Then whichever of the two is copied to their corresponding actual parameter **last** becomes the value of `p1`

3. Pass-by-Value-Result

- It is an implementation model for **inout-mode** parameters in which actual values are **copied**
- It is a combination of pass-by-value and pass-by-result
- The value of the actual parameter is used to **initialize** the corresponding formal parameter, which then acts as a local variable
- At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.
- It is sometimes called **pass-by-copy** because the actual parameter is copied to the formal parameter at subprogram entry and then copied back at subprogram termination.

4. ***Pass-by-Reference***

- *Pass-by-reference* is a second implementation model for **inout-mode** parameters
- Rather than copying data values back and forth. This method transmits an **access path**, usually just **an address**, to the called subprogram. This provides the access path to the cell storing the actual parameter
- The actual parameter is **shared** with the called subprogram
- Advantages
  - The passing process is **efficient** in terms of time and space. Duplicate space is not required, nor is any copying
- Disadvantages
  - Access to the formal parameters will be **slower** than pass-by-value, because of additional level of **indirect addressing** that is required
  - Inadvertent and erroneous **changes** may be made to the actual parameter
  - **Aliases** can be created as in C++

```
void fun(int &first, int &second)
```

  - If the call to fun happens to pass the same variable twice, as in

```
fun(total, total)
```

  - Then first and second in fun will be aliases

5. **Pass-by-Name**

- The method is an **inout-mode** parameter transmission that does not correspond to a single implementation model
- When parameters are passed by name, the actual parameter is, in effect, **textually** substituted for the corresponding formal parameter in all its occurrences in the subprogram
- A formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.
- Because pass-by-name is **not** part of any widely used language, it is not discussed further here

(3)

The subprogram Sub is called from main with the call Sub (w, x, y, z), where w is passed-by-value, x is passed-by-result, y is passed-by-value-result and z is passed-by-reference. Design one possible implementation of the common parameter-passing methods.

9.5.3 Implementing Parameter-Passing Methods

- In most contemporary languages, parameter communication takes place through the **run-time stack**
- The run-time stack is initialized and maintained by the run-time system, which is a system program that manages the execution of programs
- The run-time stack is used extensively for subprogram control linkage and parameter passing
- **Pass-by-value** parameters have their values copied into stack locations
  - The stack location then serves as storage for the corresponding formal parameters.
- **Pass-by-result** parameters are implemented as the opposite of pass-by-value
  - The values assigned to the pass-by-result actual parameters are placed in the stack, where they can be retrieved by the calling program unit upon termination of the called subprogram
- **Pass-by-value-result** parameters can be implemented directly from their semantics as a combination pf pass-by-value and pass-by-result
  - The stack location for the parameters is initialized by the call and it then used like a local variable in the called subprogram
- **Pass-by-reference** parameters are the simplest to implement.
  - Only its address must be placed in the stack
  - Access to the formal parameters in the called subprogram is by indirect addressing from the stack location of the address
- The subprogram `sub` is called from `main` with the call `sub(w, x, y, z)`, where `w` is **passed-by-value**, `x` is **passed-by-result**, `y` is **passed-by-value-result**, and `z` is **passed-by-reference**

Function call in `main`: `sub( w, x, y, z)`
Function header: `void sub(int a, int b, int c, int d)`
(pass `w` by **`value`**, `x` by **`result`**, `y` by **`value-result`**, `z` by **`reference`**)

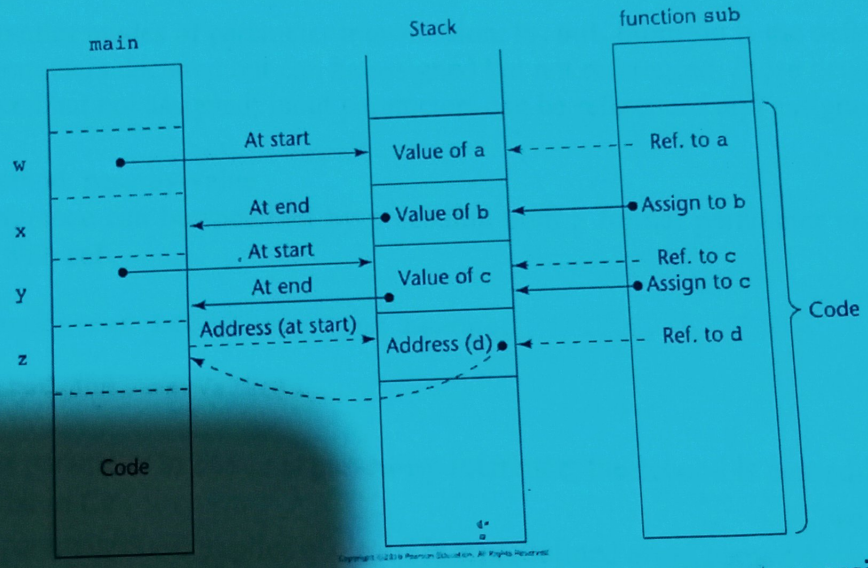


Figure 9.2 One possible stack implementation of the common parameter-passing methods

9.5.4 Parameter-Passing Methods of Some Common Languages

- Fortran
  - Always used the inout semantics model
  - Before Fortran 77: pass-by-reference
  - Fortran 77 and later: scalar variables are often passed by value-result
- C
  - Pass-by-value
  - Pass-by-reference is achieved by using **pointers** as parameters
- C++
  - A special pointer type called **reference** type. Reference parameters are **implicitly dereferenced** in the function or method, and their semantics is pass-by-reference
  - C++ also allows reference parameters to be defined to be **constants**. For example, we could have

```
void fun(const int &p1, int p2, int &p3)  { . . . }
```

    - `p1` is pass-by-reference: `p1` **cannot** be changed in the function `fun`
    - `p2` is pass-by-value
    - `p3` is pass-by-reference
    - Neither `p1` nor `p3` need be explicitly dereference in `fun`
- Java
  - All parameters are passed are **passed by value**
  - However, because objects can be accessed only through reference variables, object parameters are in effect **passed by reference**
    - Although an object reference passed as a parameter cannot itself be changed in the called subprogram, the referenced object can be changed if a method is available to cause the change
- Ada
  - Three semantics modes of parameter transmission: **in, out, inout**; in is the default mode
  - Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; inout parameters can be referenced and assigned
- C#
  - Default method: pass-by-value
  - Pass-by-reference can be specified by preceding both a formal parameter and its actual parameter with **ref**

```
void sumer(ref int oldSum, int newOne)  { . . . }
. . .
sumer(ref sum, newValue);
```

    - The first parameter to `sumer` is passed-by-reference; the second is passed-by-value
- PHP: very similar to C#
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`

- Python and Ruby use pass-by-assignment (all data values are objects)
  - The process of changing the value of a variable with an assignment statement, as in

```
x = x + 1
```

  - does not change the object referenced by `x`. Rather, it takes the object referenced by `x`, increments it by `1`, thereby creating a **new** object (with the value `x + 1`), and then `x` to reference the new object.

9.5.5 Type-Checking Parameters

- It is now widely accepted that software reliability demands that the types of actual parameters be checked for consistency with the types of the corresponding formal parameters.
- Ex:

```
result = sub1(1)
```

  - The actual parameter is an integer constant. If the formal parameter of `sub1` is a floating-point type, no error will be detected without parameter type checking.
- Early languages, such as Fortran 77 and the original version of C, did **not** require parameter type checking
- Pascal, **Java**, and Ada: it is always **required**
- Relatively new languages Perl, JavaScript, and PHP do **not** require type checking

9.6 Parameters That Are Subprograms 393

- The issue is what referencing environment for executing the passed subprogram should be used
- The three choices are:
  1. **Shallow binding:** The environment of the call statement that enacts the passed subprogram
     - Most natural for **dynamic-scoped** languages
  2. **Deep binding:** The environment of the definition of the passed subprogram
     - Most natural for **static-scoped** languages
  3. **Ad hoc binding:** The environment of the call statement that passed the subprogram as an actual parameter
     - It has **never** been used because, one might surmise, the environment in which the procedure appears as a parameter has no natural connection to the passed subprograms
- Ex: JavaScript

```
function sub1() {
    var x;
    function sub2() {
        alert(x);       // Creates a dialog box with the value of x
    };
    function sub3() {
        var x;
        x = 3;
        sub4(sub2);
    };
    function sub4(subx) {
        var x;
        x = 4;
        subx();
    };
    x = 1;
    sub3();
};
```

- Consider the execution of **sub2** when it is called in `sub4`.
  - Shallow binding: the referencing environment of that execution is that of `sub4`, so the reference to `x` in `sub2` is bound to the local `x` in `sub4`, and the output of the program is **4**.
  - Deep binding: the referencing environment of `sub2`'s execution is that of `sub1`, so the reference so the reference to `x` in `sub2` is bound to the local x in `sub1` and the output is **1**.
  - Ad hoc binding: the binding is to the local `x` in `sub3`, and the output is **3**.

9.7 Calling Subprograms Indirectly 395

- There are situations in which subprograms must be called indirectly
- These most often occur when the specific subprogram to be called is not known until **run time**
- The call to the subprogram is made through a pointer or reference to the subprogram, which has been set during execution before the call is made
- C and C++ allow a program to define a **pointer to function**, through which the function can be called
- For example, the following declaration defines a pointer (`pfun`) that can point to any function that takes a `float` and an `int` as parameters and returns a `float`

```
float (*pfun) (float, int);
```

- **Both** following are legal ways of giving an initial value or assigning a value to pointer to a function

```
int myfun2(int, int); // A function declaration
int (*pfun2)(int, int) = myfun2; // Create a pointer and initialize
                                 // it to point to myfun2

int myfun2(int, int); // A function declaration
pfun2 = myfun2;       // Assigning a function's address to a pointer
```

  - The function `myfun2` can now be called with either of the following statements:

```
(*pfun2)(first, second);

pfun2(first, second);
```

  - The first of these explicitly dereferences pointer `pfun2`, which is legal, but **unnecessary**

- In C#, the power and flexibility of method pointers is increased by making them **objects**
- These are called **delegates**, because instead of calling a method, a program delegates that action to a delegate
- For example, we could have the following:

```
public delegate int Change(int x);
```

  - This delegate type, named `Change`, can be instantiated with any method that takes an `int` as a parameter and returns an `int`
  - For example, consider the following method:

```
static int fun1 (int x) {. . .}
```

  - The delegate change can be instantiated by sending the name of this method to the delegate's constructor, as in the following:

```
Change chgfun1 = new Change(fun1);
```

  - This can be shortened to the following:

```
Change chgfun1 = fun1;
```

  - Following is an example call to `fun1` through the delegate `chgfun1`:

```
chgfun1(12);
```

  - Objects of a delegate class can store more than one method. A second method can be added using the operator +=, as in the following

```
Change chgfun1 += fun2;
```

- Ada 95 has pointers to subprograms, but Java does **not**

9.8 Design Issues for Functions

- The following design issues are specific to functions:
  - Are side effects allowed?
  - What types of values can be returned?
  - How may values can be returned?

- Functional side effects
  - Because of the problems of side effects of functions that are called in expressions, parameters to functions should always be **in-mode** parameters
  - For example, Ada functions can have only in-mode formal parameters
  - This effectively prevents a function from causing side effects through its parameters or through aliasing of parameters and globals
  - In most languages, however, functions can have either **pass-by-value** or **pass-by-reference** parameters, thus allowing functions that cause side effects and aliasing

- Types of returned values
  - Most imperative programming languages restrict the types that can be returned by their functions
  - C allows **any** type to be returned by its functions **except** arrays and functions
  - C++ is like C but also allows user-defined types, or classes, to be returned from its functions
  - Java and C# methods can return any type (but because methods are not types, methods cannot be returned)
  - Python, Ruby, and Lua treat methods as first-class objects, so they can be returned, as well as any other class
  - JavaScript functions can be passed as parameters and returned from functions

- Number of return values
  - In most of languages, only a **single** value can be returned from a function
  - Ruby allows the return of more than one value from a method
  - Lua also allows functions to return **multiple** values
    - Such values follow the return statement as a comma-separated list, as in the following:

```
return 3, sum, index
```

    - If the function returned three values and all are to be kept by the caller, the function would be called as in the following example:

```
a, b, c = fun()
```

  - In F#, multiple values can be returned by placing them in a tuple and having the tuple be the last expression in the function

- An overloaded operator is one that has multiple meanings
- The meaning of a particular instance of an overloaded operator is determined by its types of its operands
- For example, if the * operator
  - It has two floating-point operands in a Java program, it specifies floating-point multiplication
  - But if the same operator has two integer operands, it specifies integer multiplication
- An overloaded subprogram is a subprogram that has **the same name** as another subprogram in the same referencing environment
- Every version of an overloaded subprogram must have a unique protocol; that is, it must be different from the others in the number, order, or types of its **parameters**, or in its return if it is a function
- The meaning of a call to an overloaded subprogram is determined by the actual parameter list
- Ada, Java, C++, and C# include predefined overloaded subprograms
  - For examples, overloaded constructors
  - Users are also allowed to write multiple versions of subprograms with the same
- Overloaded subprograms that have default parameters can lead to **ambiguous** subprogram calls

```
void fun(float b = 0.0);
void fun( );
. . .

fun( ); // The call is ambiguous and will cause a compilation error
```

9.10 Generic Subprograms 400

- A generic or polymorphic subprogram takes parameters of **different types** on different activations
- Overloaded subprograms provide a particular kind of polymorphism called ad hoc polymorphism
- Subtype polymorphism means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- Parametric polymorphism is provided by a subprogram that takes a generic parameter that is used in a type expression that describes the types of the parameters of the subprogram

Generic Functions in C++

- Generic functions in C++ have the descriptive name of **template** functions
  - Generic subprograms are preceded by a template clause that lists the generic variables, which can be type names or class names

```
template <class Type>
Type max(Type first, Type second) {
   return first > second ? first : second;
}
```

    - where `Type` is the parameter that specifies the type of data on which the function will operate
  - For example, if it were instantiated with `int` as the parameter, it would be:

```
int max(int first, int second) {
   return first > second ? first : second;
}
```

  - The following is the C++ version of the generic sort subprogram

```
template <class Type>
void generic_sort (Type list [ ], int len) {
   int top, bottom;
   Type temp;
   for (top = 0, top < len - 2; top ++)
        for (bottom = top + 1; bottom < len - 1; bottom++)
   if (list [top] > list [bottom]) {
        temp = list [top];
        list[top] = list[bottom];
   } // end for bottom
} // end for generic
```

  - The instantiation of this template function is:

```
float flt_list [100];
. . .
generic_sort (flt_list, 100);
```

Generic Methods in Java 5.0

- Java's generic methods differ rom the generic subprogram of C++ in serval important ways:
  - Generic parameters in Java 5.0 must be **classes** – they cannot be primitive type
  - Java 5.0 generic methods are instantiated just once as truly generic methods
  - Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters. Such restrictions are called bounds
- As an example of a generic Java 5.0 method:

```
public static <T> T doIt(T[] list) { . . . }
```

  - This defines a method named `doIt` that takes an array of elements of a generic type
  - The name the generic type is `T` and it must be an array
  - An example call to `doIt`:

```
doIt<String>(myList);
```

  - Generic parameters can have bounds:

```
public static <T extends Comparable> T doIt(T[] list) { . . . }
```

    - The generic type must be of a class that implements the `Comparable` interface

Generic Methods in C# 2005

- The generic method of C# 2005 are similar in capability to those of Java 5.0
- One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
- For example, consider the following skeletal class definition:

```
class MyClass {
   pulbic static T DoIt<T>(T p1) { . . . }
   . . .
}
```

  - For example, both following class are legal:

```
int myInt = MyClass.DoIt(17);           // Calls DoIt<int>
string myStr = MyClass.DoIt('apples'); // Calls DoIt<string>
```

9.11 User-Defined Overloaded Operators 406

- Operators can be overloaded in Ada, C++, Python, and Ruby (**not** carried over into Java)
- A Python example:

```
def __add__ (self, second) :
  return Complex(self.real + second.real,
                 self.imag + second.imag)
```

  - The method is named __add__

- For example, the expression `x + y` is implemented as

```
x.__add__(y)
```

9.12 Closures 406

- A closure is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit **nested** subprograms does not need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

- Following is an example of a closure written in JavaScript:

```
function makeAdder(x) {
   return function(y) {return x + y;}
}
. . .
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("add 10 to 20: " + add10(20) + "<br />");
document.write("add 5 to 20: " + add5(20) + "<br />");
```

  - The closure is the anonymous function returned by `makeAdder`
  - The output of this code, assuming it was embedded in an HTML document and displed with a browser, is as follows:

```
Add 10 to 20: 30
Add 5 to 20: 25
```

  - In this example, the closuer is the **anonymous** function defined inside the `makeAdder` function, which `makceAdder` return
  - The variable `x` referenced in the closure function is bound to the parameter what was sent to `makeAdder`
  - The `makeAdder` function is called twice, once with a parameter of `10` and once with `5`
  - Each of these call returns a different version of the closure because they are bound to different values of `x`
  - The first call to `makeAdder` creates a function that adds `10` to it parameter; the second creates a function that adds `5` to its parameter

- In C#, this same closure function can be written in C# using nested anonymous delegate
  - The type of the nesting method is specified to be a function that takes an `int` as a parameter and returns an anonymous **delegate**
  - The return type is specified with the special nation for such delegates, `Func<int, int>`
    - The first type in the angle brackets is the parameter type
    - The second type is the return type of the method encapsulated by the delegate

```
static Func<int, int> makeAdder(int x) {
   return delegate(int y) {return x + y;};
}
. . .
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}", Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

    - The output of this code is exactly the same as for the previous JavaScript closure example

```
Add 10 to 20: 30
Add 5 to 20: 25
```

9.13 Coroutines 408

- A coroutine is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- The coroutine control mechanism is often called the symmetric control: caller and called coroutines are more equitable
- It also has the means to maintain their status between activation
- This means that coroutines must be **history sensitive** and thus have static local variables
- Secondary executions of a coroutine often begin at points other than its beginning
- The invocation of a coroutine is named a **resume** rather than a call
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasi-concurrent execution of program units (the coroutines); their execution is interleaved, but not overlapped

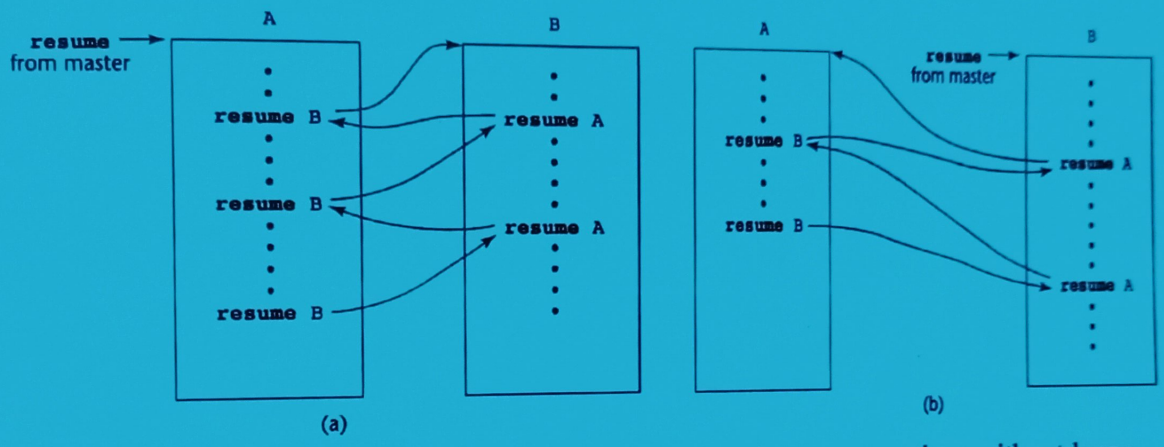


Figure 9.3 Two possible execution control sequences for two coroutines without loops

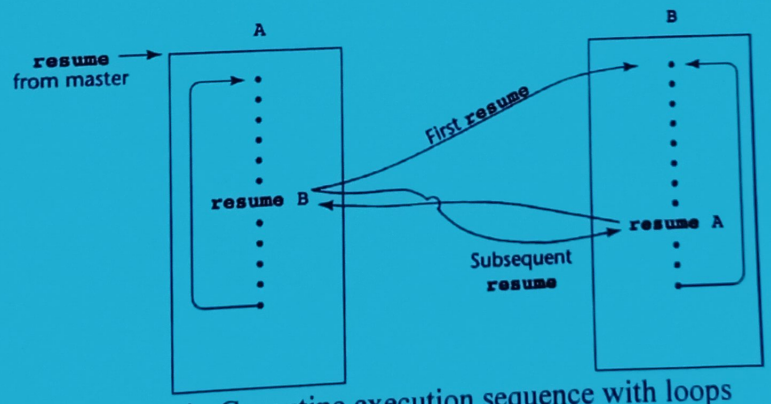


Figure 9.4 Coroutine execution sequence with loops

Summary 411

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
  - Functions return values and procedures do **not**
- Local variables in subprograms can be stack-dynamic or static
- JavaScript, Python, Ruby, and Lua allow subprogram definitions to be **nested**
- Three models of parameter passing: **in**-mode, **out**-mode, and **inout**-mode
- Five implementation models of parameter passing:
  - **Pass-by-value**: in-mode
  - **Pass-by-result**: out-mode
  - **Pass-by-value-result**: inout-mode
  - **Pass-by-reference**: inout-mode
  - **Pass-by-name**: inout-mode
- C and C++ support **pointers to functions**. C# has **delegates**, which are object that can store references to methods
- Ada, C++, C#, Ruby, and Python allow both subprogram and **operator overloading**
- Subprograms in C++, Java 5.0, and C# 2005 can be **generic**, using parametric polymorphism, so the desired types of their data objects can be passed to the **compiler**, which then can construct unit for the required types
- A closure is a subprogram and its reference environment
  - Closures are useful in languages that allow **nested** subprograms, are **static-scoped**, and allow subprograms to be returned from functions and assigned to variables
- A coroutine is a special subprogram with multiple entries