Chapter 7
Expressions and Assignment Statements

7.1 Introduction 302
7.2 Arithmetic Expressions 302
7.3 Overloaded Operators 311
7.4 Type Conversions 313
7.5 Relational and Boolean Expressions 316
7.6 Short-Circuit Evaluation 318
7.7 Assignment Statements 319
7.8 Mixed-Mode Assignment 324
Summary • Review Questions • Problem Set • Programming Exercises 324
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:

<table>
<thead>
<tr>
<th></th>
<th><strong>Ruby</strong></th>
<th><strong>C-Base Languages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Highest</strong></td>
<td>**</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>unary +, -</td>
<td>prefix ++, --, unary +, -</td>
</tr>
<tr>
<td></td>
<td>*, /, %</td>
<td>*, /, %</td>
</tr>
<tr>
<td><strong>Lowest</strong></td>
<td>binary +, -</td>
<td>binary +, -</td>
</tr>
</tbody>
</table>
• **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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Associativity Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby, Fortran</td>
<td>Left: *, /, +, -</td>
</tr>
<tr>
<td></td>
<td>Right: **</td>
</tr>
<tr>
<td>C-based languages</td>
<td>Left: *, /, %, binary +, binary -</td>
</tr>
<tr>
<td></td>
<td>Right: ++, --, unary -, unary +</td>
</tr>
</tbody>
</table>

  – **APL** is different; all operators have equal precedence and all operators associate **right to left**.
  – Ex: APL
    
    \[ A \times B + C \]  // A = 3, B = 4, C = 5
    
    \[ \Rightarrow 27 \]
  – 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:
    
    \[(A + B) \times C\]  // addition will be evaluated first
• Expressions in Lisp
  – All arithmetic and logic operations are by explicitly called subprograms
  – Ex: to specify the c expression \( a + b \cdot c \) in Lisp, one must write the following expression:

\[
(+ \ a \ (* \ b \ c)) \quad // \quad \text{+ and \ *} \ are \ the \ names \ of \ \text{functions}
\]

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

\[
\text{if} \ (\text{count} == 0) \\
\quad \text{average} = 0; \\
\text{else} \\
\quad \text{average} = \text{sum} / \text{count};
\]

  – 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).

\[
\text{expression}_1 \ ? \ \text{expression}_2 : \ \text{expression}_3
\]

  – Ex:

\[
\text{average} = (\text{count} == 0) \ ? \ 0 : \text{sum} / \text{count};
\]
7.2.2 Operand evaluation order

- Operand evaluation order:
  - Variables: fetch the value from 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 illustrates 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++)

\[
x = &y \quad // & \text{ as } \textbf{unary} \text{ operator is the address of } y
\]
\[
// & \text{ as } \textbf{binary} \text{ operator bitwise logical AND}
\]
- 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
7.4 Type Conversions 313

- 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
      ```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 was typed as `a`
      - Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. Simply, `a` 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
    ```java
    (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

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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Ada</th>
<th>C-Based Languages</th>
<th>Fortran 95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>=</td>
<td>==</td>
<td>.EQ. or ==</td>
</tr>
<tr>
<td>Not Equal</td>
<td>/=</td>
<td>!=</td>
<td>.NE. or &lt;&gt;</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>.GT. or &gt;</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>.LT. or &lt;</td>
</tr>
<tr>
<td>Greater than or equal</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td>.GE. or &gt;=</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>&lt;=</td>
<td>&lt;=</td>
<td>.LE. or &gt;=</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>FORTRAN 77</th>
<th>FORTRAN 90</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>and</td>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>.OR.</td>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NOT.</td>
<td>not</td>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>

- 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 \(0 \times x = 0\) 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 |)
7.7 Assignment Statements 319

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:

  ```c
  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

  ```c
  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.

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

  instead of

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

Multiple Assignments

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

  ```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**:

  ```perl
  ($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:
  
  ```ml
  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
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:

```c
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 lowering reliability