Chapter 7

Expressions and Assignment Statements

Topics

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

- Expressions are the fundamental means of specifying *computations* in a programming language.
- To understand expression evaluation, we need to be familiar with the *orders* of operator and operand *evaluation*.
- Essence of imperative languages is a dominant role in assignment statements.
- Recall: chap.3

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expr>
<var> → A | B | C
<expr> → <var> + <var>
       | <var> - <var>
       | <var>
```

Arithmetic Expressions (AEs)

- One of the motivations for the development of the first programming languages.
- Arithmetic Expressions consist of *operators*, *operands*, *parentheses*, and *function calls*.
- In most languages, binary operators are *infix:*

  operand1 *operator* operand2
- In Scheme and LISP, binary operators are prefix:

  operator operand1 operand2
- Perl also has *some* prefix binary operators.
- Most *unary* operators:
  - Most of them: *prefix*, e.g.) –a
  - ++ and -- operators in C-based languages: either prefix or postfix.

Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - Operator *precedence* rules?
  - Operator *associativity* rules?
  - Order of *operand evaluation*?
  - Operand evaluation *side effects*?
  - Operator *overloading*?
  - *Type mixing* in expressions?

Arithmetic Expressions: Operators

- A *unary* operator: *one* operand
  - E.g.) $A \vee (\neg B)$, $2 * (-3)$
- A *binary* operator: *two* operands:
  - E.g.) $A \wedge B$, $2 + 3$
- A ternary operator: three operands
  - E.g.) average = $(cout == 0)_1$? 0_2 : $sum/count_3$

Arithmetic Expressions: Operator *Precedence* Rules

- The *operator precedence rules* for expression evaluation: defines the order of evaluation for the "*adjacent*" operators of *different precedence levels*.
- Typical precedence levels
  1. parentheses
  2. unary operators
  3. ** (exponent, if the language supports it)
  4. *, /
  5. +, –

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

Arithmetic Expressions: Operator *Precedence* Rules postfix vs. prefix

```c
#include <stdio.h>

int main() {
   int var1 = 5, var2 = 5;

// postfix
   // the current value of var1, 5 is used for display.
   // Then, var1 is increased to 6.
   printf("%d\n", var1++);
//   var1++;
   printf("%d\n", var1);
   var1 =+ 3; // NO postfix assignment is used for var1 != var1 + 3,  var1 = 0 + 3.
   printf("%d\n", var1);

   printf("\n");


// prefix
   // var2 is increased to 6
   // Then, it is displayed.
   printf("%d\n", ++var2);

   var2 += 10;    // prefix assignment is used for var2 = var2 + 10
   printf("%d\n", var2);
   var2 += var1;    // prefix assignment: var2 = var2 + var1
   printf("%d\n", ++var2);
   return 0;
}
```

postfix

```
5
6
3
```

prefix

```
6
16
20
```

Arithmetic Expressions: Operator *Associativity* Rule

- The *operator associativity rules* for expression evaluation: define the order of evaluation for the adjacent operators with the *same precedence level*.
- Typical associativity rules
  - Left associativity: Left to right, e.g.) a-b+c = (a-b)+c
  - Exception: ** - right associativity, e.g.) 2**3**4 = 2**(3**4)=$2^{(3^4)}$
  - Sometimes unary operators associate right to left

    (e.g., in FORTRAN)
- In APL: all operators have *equal precedence* and

  all operators associate *right to left*.
- Precedence and associativity rules can be overridden with parentheses.

Expressions: in Ruby and Scheme

- Ruby
  - All arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as *methods*.
  - E.g.) `a + b`: a call to the + method of the object referenced by `a`, passing the object referenced by `b` as a parameter. i.e. a.+(b)
- Scheme (and Common Lisp)
  - All arithmetic and logic operations are explicitly called *subprograms.*
  - `a + b * c` is coded as `(+ a (* b c))` – prefix

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)

```
exp_1 ? exp_2 : exp_3
```

  - Example:

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

  - Evaluates as if written as follows:

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

Arithmetic Expressions: Operand Evaluation Order

- *Operand Evaluation Order*: if operand is
  1. Variables: fetch the value from memory
  2. Constants:
     - sometimes a fetch from memory;
     - sometimes the constant is in the machine language instruction.
  3. *Parenthesized* expressions: all of the operators it contains must be evaluated before its value can be used as an operand.
  4. The potential side effects when an operand is a function call: e.g.) `a + fun(a)`

Arithmetic Expressions: Potentials for Side Effects

- *Functional side effects:* when a function changes a two-way parameter or a non-local variable.
- Problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression (two-way parameters)
  - e.g., for a parameter change:

```
int a = 10;
fun(a){
      a = a + 10; /* parameter a is changed */
      return 5
}
void main() {
      b = a + fun(a);     → a = 20
}     → If  a     is evaluated before fun(a), b = 15
      → If fun(a) is evaluated before a,      b = 25
```

Functional Side Effects (cont.)

- Solutions:
  1. Write the language definition to *disallow* functional side effects.
     - No two-way parameters in functions
     - No non-local references in functions
     - **Advantage:** it works!
     - **Disadvantage:** inflexibility of one-way parameters and lack of non-local references - time for parameter passing
  2. Write the language definition to demand that operand *evaluation order* be *fixed*.
     - **Disadvantage**: limits *some* compiler optimizations – need to reorder operand evaluation.
     - Java requires that operands appear to be evaluated in left-to-right order.

Referential Transparency

- A program has the property of *referential transparency* if any two expressions that have the same value can be *substituted* for one another anywhere in the program, without affecting the action of the program.

```
result1 = (fun(a) + b) / (fun(a) – c);
temp = fun(a);
result2 = (temp + b) / (temp – c);
```

If `fun` has no side effects, `result1 = result2`

Otherwise, referential transparency is violated.

e.g.) `fun` has the side effect of changing a value of `b` or `c`

→ `result1` ≠ `result2`

Referential Transparency (cont.)

- Advantage of referential transparency.
  - Semantics of a program is much easier to understand

    if it has referential transparency.
- Programs in *pure functional languages* are referentially transparent because they have no variable. – chap.15
  - Functions cannot have a state, which would be stored in local variables.
  - If a function uses an outside value, it must be a constant (there are no variables).

    So, the value of a function depends only on its parameters.

Overloaded Operators

- *Operator overloading:*

  The use of an operator for more than one purpose.
- e.g.) `+` for `int` and `float` – common.
- some potential trouble: e.g.) `&` in C and C++
  - `&` : a *binary* operator as a *bitwise logical AND* operator

    vs. a *unary* operator as the *address-of* operator

    e.g.) `x = &y`
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Loss of readability using the same symbol for two completely unrelated operations.

Overloaded Operators (cont.)

- *User-defined* overloaded operators: in C++, C#, F#
  - When sensibly used, they aid readability
    (avoid method calls, expressions appear natural)
  - E.g.) In matrix ADT,

    `MatrixAdd(MatrixMult(A,B), MatrixMult(C,D))`

→ `A * B + C * D` with the user-defined operator overloading for a matrix abstract data type.

  - Potential Problems:
    - Users can define nonsense operations
    - Readability may suffer if different groups overload the same operators in different ways (even when the operators make sense)

Type Conversions

- *Narrowing conversion:*

  *C*onversion of an object to a type that cannot include all of the values of the original type.

  – e.g., **`float`** to **`int`**

- *Widening conversion:*

  Conversion of an object to a type that can include at least approximations to all of the values of the original type.

  – e.g., **`int`** to **`float`**

Type Conversions: Mixed Mode

- *Mixed-mode expression:*
  with the operands of *different types*.
- *Coercion:*
  - an *implicit* type conversion.
  - E.g.) `a+b` where `a` is `int` and `b` is `float` → coercion of `a` to `float` in computation.
  - Disadvantage of coercions:
    - They decrease 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#: no coercions in expressions.

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - C: `(int)angle`
  - F#: `float(sum)`
  - Python:
    ```
    a = input()
    c = 5 + int(a)
    ```
    - explicit conversion of input string in `a` to an integer.

In F# and Python, the syntax of casting is similar to that of function calls.

Errors in Expressions

- Causes
  - Inherent limitations of arithmetic.
    - e.g.) division by zero
  - Limitations of computer arithmetic.
    - e.g.) overflow, underflow
  - The above cases are *run-time errors*, called *exceptions.*
  - Language facilities that allow programs to detect and deal with exceptions.

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types.
  - Evaluate to some *Boolean* representation.
  - Operator symbols used vary somewhat among languages.
    E.g.) `(!=, /=, ~=, .NE., <>, #)` - inequality
- JavaScript and PHP have two additional relational operator:
  `===` and `!==`
  - Similar to `==` and `!=`, but NO *coercion of* their operands.
  - E.g.) `"7" == 7` → true, `"7" === 7` → false
  - Ruby: `==` for equality relation operator using coercion
    vs. `eql?` for those without using coercion.

Relational and Boolean Expressions (cont.)

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
- C89: 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,

  - Expected result: **`a < b and b < c`**
  - Actual result: **`(a < b) < c`**
    - Left operator is evaluated, producing 0 (false) or 1 (true)
    - Then, the result is compared with the third operand (i.e., **`c`**)
      - i.e. `0 < c` or `1 < c`

Short Circuit Evaluation

- The result of an expression is determined *without* evaluating all of the operands and/or operators.
  - An early decision on the result.
- Example: `(13 * a) * (b / 13 – 1)`

  If `a` is zero, there is no need to evaluate `(b/13-1)`
- Problem with *non*-short-circuit evaluation

```
index = 0;
while (index < length) && (LIST[index] != value)
      index++;
```

  - When `index=length, LIST[index]` will cause an indexing error (assuming `LIST[0..length – 1]` long)

Short Circuit Evaluation (cont.)

- C, C++, and Java:
  - the usual Boolean operators (&& and ||): use short-circuit evaluation
  - bitwise Boolean operators (& and |): not short circuit
- Ruby, Perl, ML, F#, and Python:
  - All logic operators are short-circuit evaluated.
- Short-circuit evaluation exposes the potential problem of side effects in expressions: e.g) `(a > b) || (b++ / 3)`
  - b is changed (in the 2^{nd} arithmetic expression) only when $a \le b$.
  - If the programmer assumed b would be changed every time this expression is evaluated during execution (and the program's correctness depends on it), the program will fail.

Assignment Statements

- The general syntax

  `<target_var> <assign_operator> <expression>`
- The assignment operator

  `=` Fortran, BASIC, the C-based languages

  `:=` Ada
- `=` can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use `==` as the relational operator)

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

```
($flag ? $total : $subtotal) = 0
```

which is equivalent to

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```

- Cf) Conditional expression (#11)

```
exp_1 ? exp_2 : exp_3
```

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

Assignment Statements: Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL

→ adopted by C and the C-based languages

  - Example

```
a = a + b
```

can be written as

```
a += b
```

-- prefix assignment

Assignment Statements: Unary Assignment Operators (UAO)

- UAOs in C-based languages combine increment and decrement operations with assignment.
- Examples:

`sum = ++count:`

`count` incremented, then assigned to `sum`

i.e. `count=count+1; sum = count;`

`sum = count++:`

`count` assigned to `sum`, then incremented

i.e. `sum = count; count=count+1;`

`count++ : count` incremented

`-count++ : count` incremented then negated (ref. #7)

Assignment as an Expression

- In the C-based languages, Perl, and JavaScript:
  the assignment statement produces a *result* and
  can be used as an *operand*.

  ```
  while ((ch = getchar())!= EOF){…}
  ```

  `ch = getchar()` is carried out;
  the result (assigned to `ch`) is used as a conditional value for the `while` statement
- Example: `a = b +2 (c=d/b)1 -3 1;`
- multiple-target assignment: `sum = count = 0;`
  - `count` is assigned the zero → `count`'s value is assigned to sum.
- Disadvantage: another kind of expression side effect.

Multiple Assignments

- Perl and Ruby: multiple-target & multiple-source assignments are allowed.

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

Also, the following is legal and performs an interchange:

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

is equivalent to

```
$temp = $first
$first = $second
$second = $first
```

Assignment in Functional Languages (chap.15)

- Identifiers in functional languages are only names of values – not a variable.
- ML
  - Names are bound to values with `val`: value declaration

    ```
    val fruit = apples + oranges;
    ```

  - If another `val` for `fruit` follows, it is a new and different name where the previous version of `fruit` is hidden.
- F#
  - Name binding to values with `let`.
  - `Let` is like ML's `val`, except `let` also creates a new scope.

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode.
- In Fortran, C(?), Perl, and C++: any numeric type value can be assigned to any numeric type variable.

```c
int main() {
  // Create variables
  int myNum = 5;              // Integer (whole number)
  float myFloatNum = 5.99;    // Floating point number
  char myLetter = 'D';        // Character

  // Print variables
  printf("%d\n", myNum);
  printf("%f\n", myFloatNum);
  printf("%c\n", myLetter);
  printf("\n");

  myNum = 10.5;
  printf("%d\n", myNum);
  printf("%f\n", myNum);
  printf("\n");

  myFloatNum = 9;
  printf("%d\n", myFloatNum);
  printf("%f\n", myFloatNum);
}
```

```
5
5.990000
D

10
0.000000

1
9.000000
```

```
5
5.990000
D

10
0.000000

31924896
9.000000
```

- In Java and C#: only widening assignment coercions are done.
- In Ada: no assignment coercion.

Summary

- Expressions
- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment