Expression Evaluation and Control Flow

In Text: Chapter 6

Outline

- Notation
- Operator Evaluation Order
- Operand Evaluation Order
- Overloaded operators
- Type conversions
- Short-circuit evaluation of conditions
- Control structures

Arithmetic Expressions

- Design issues for arithmetic expressions
  - Notation form?
  - 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?

Operators

- A **unary** operator has one operand
- A **binary** operator has two operands
- A **ternary** operator has three operands
- Functions can be viewed as unary operators with an operand of a simple list

Operators

- Argument lists (or parameter lists) treat separators (comma, space) as "stacking" or "append" operators
- A keyword in a language statement can be viewed as functions in which the remainder of the statement is the operand

Notation & Placement

- Prefix
  - **op** a b **op**(a,b) (**op** a b)
- Infix
  - a **op** b
- Postfix
  - a b **op**

Notation & Placement

- Most imperative languages use infix notation for binary and prefix for unary operators
- Lisp: prefix
  - (op a b)

Operator Evaluation Order [1]

- Precedence
- Associativity
- Parentheses

Operator Precedence

- Define the order in which "adjacent" operators of different precedence levels are evaluated
  - Parenthetical groups (...)
  - Exponentiation **
  - Mult & Div * , /
  - Add & Sub + , -
  - Assignment :=
- Where to put the parentheses?
  - E.g., A * B + C ** D / E - F

Operator Associativity

- Define the order in which adjacent operators with the same precedence level are evaluated:
  - Left associative * , / , + , -
  - Right associative ** (exponentiation)
- Where to put the parentheses?
  - E.g., B ** C ** D - E + F * G / H

Operator Associativity

- EFFECTIVELY
  - Most programming languages evaluate expressions from left to right
  - LISP uses parentheses to enforce evaluation order
  - APL is strictly RIGHT to LEFT, taking note only of parenthetical groups

Operator Associativity

- Associativity
  - For some operators, the evaluation order does not matter, i.e., (A + B) + C = A + (B + C)
- However, in a computer when floating-point numbers are represented approximately, the mathematical "associativity" does not always hold
  - E.g., A = 200, B = Float.MIN_VALUE, C = -10

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

Parentheses

- Advantages
  - Allow programmers to specify any desired order of evaluation
  - Do not require author or reader of programs to remember any precedence or association rules
- Disadvantages
  - Can make writing expressions more tedious
  - May seriously compromise code readability

Operand Evaluation Order

- If none of the operands of an operator has side effects, then the operand evaluation order does not matter
- What are side effects ?
- Referential transparency and side effects

Side Effects

- Often discussed in the context of functions
- A side effect is some permanent state change caused by execution of functions
- The subsequent computation is influenced other than by the return value for use
  - j = i++
  - a = 10, b = a + fun(&a) (assume the function can change its parameter value)

Side Effects

- Many imperative languages distinguish between
  - *expressions*, which always produce values, and may or may not have side effects, and
  - *statements*, which are executed solely for their side effects, and return no useful value
- Imperative programming is sometimes called "computing via side effects"

N. Meng, S. Arthur 17

Side Effects

- Pure functional languages have no side effects
  - The value of an expression depends only on the referencing environment in which the expression is evaluated, *not* the time at which the evaluation occurs
    - If an expression yields a certain value at one point in time, it is guaranteed to yield the same value at any point in time

N. Meng, S. Arthur 18

How to avoid side effects ?

- Design the language to disallow functional side effects
  - No pass-by-reference parameters in functions
  - Disallow global variable access in functions
- Concerns
  - Programmers need the flexibility to return more than one value from a function
  - Passing parameters is inefficient compared with accessing global variables

How to avoid side effects ?

- Design the language with a strictly fixed evaluation order between operands
- Concerns
  - Disallow some optimizations which involve reordering operand evaluations

Referential Transparency and Side Effects

- A program has the property of referential transparency if any two expressions having the same value can be substituted for one another

E.g., result1 = (fun(a) + b) / (fun(a) – c); ⇔
temp = fun(a);
result2 = (temp + b) / (temp - c),

given that the function fun has no side effect

Key points of referentially transparent programs

- Semantics is much easier to understand
  – Being referentially transparent makes a function equivalent to a mathematical function
- Programs written in pure functional languages are referentially transparent
- The value of a referentially transparent function depends on its parameters, and possibly one or more global constants

Overloaded Operators

- The multiple use of an operator is called operator overloading
  - E.g., "+" is used to specify integer addition, floating-point addition, and string catenation
- Do not use the same symbol for two completely unrelated operations, because that can decrease readability
  - In C, "&" can represent a bitwise AND operator, and an address-of operator

Type Conversion

- Narrowing conversion
  - To convert a value to a type that cannot store all values of the original type
  - E.g., double->float, float->int
- Widening conversion
  - To convert a value to a type that can include all values belong to the original type
  - E.g., int->float, float->double

Narrowing Conversion vs. Widening Conversion

- Narrowing conversion are not always safe
  - The magnitude of the converted value can be changed
  - E.g., float->int with 1.3E25, the converted value is distantly related to the original one
- Widening conversion is always safe
  - However, some precision may be lost
  - E.g., int->float, integers have at least 9 decimal digits of precision, while floats have 7 decimal digits of precision

25

Implicit Type Conversion

- A **coercion** is an implicit type conversion
- Arithmetic expressions with operators that can have differently typed operands are called **mixed-mode expressions**
- Languages allowing such expressions must define implicit operand type conversions

N. Meng, S. Arthur 26

Implicit Type Conversion

```
var x, y: integer;
      z: real;
      ...
y := x * z;    /* x is automatically converted to "real"  */
```

- Implicit type conversion can be achieved by narrowing or widening one or more operators
- It is better to widen when possible
  - E.g., x = 3, z = 5.9
    y = 17 if x is widened, y = 15 if z is narrowed

Key Points of Implicit Coercions

- They decrease the type error detection ability of compilers
  - Did you really mean to use "mixed-mode expressions" ?
- In most languages, all numeric types are coerced in expressions, using widening conversions

Explicit Type Conversion

- Also called "casts"
- Ada example

```
FLOAT(INDEX)-- INDEX is an INTEGER
```

- C example:

```
(int) speed /* speed is a float */
```