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

- 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

- Only Fortran, Ruby, Visual Basic, and Ada have the exponentiation operator. In all four, exponentiation operator has higher precedence than unary operators
  – Where to place the parentheses in -A**B?

- The precedence of the arithmetic operators of Ruby and the C-based languages (e.g., C, C++, Java, Python)

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

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 peers without parentheses

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

- Although we need parentheses in infix expressions, we don't need parentheses in prefix and postfix expressions
  - The operators are no longer ambiguous with respect to the operands that they work on in prefix and postfix expressions

Expression Conversion

Infix Expression	Prefix Expression	Postfix Expression
A+B	+ A B	A B +
A+B*C	?	?
(A+B)*C	?	?

A Motivating Example

- What is the value of the following expression?
  3 10 + 4 5 - *

How do you automate the calculation of a postfix expression ?

- Assuming operators include:
  Highest * /
  Lowest binary + -
- Input: a string of a postfix expression
- Output: a value
- Algorithm ?

Project 1

- Create an evaluator for logical expressions written in postfix notation
- Assuming operators include:

Highest	not "!"	RIGHT associative
	and "&"	LEFT associative
	not equal "/", equal "="	LEFT associative
Lowest	or "|"	LEFT associative

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"

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

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

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

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, what is y's value if x is widened? How about z 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 */
```

Short-Circuit Evaluation

- A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators
  - Consider `(a < b) && (b < c)`:
    - If `a >= b`, there is no point evaluating `b < c` because `(a < b) && (b < c)` is automatically false
- (x && y) ≡ if x then y else false
- (x || y) ≡ if x then true else y

Short-Circuit Evaluation

- Short-circuit evaluation may lead to unexpected side effects and cause error
  - E.g., (a > b) || ((b++) / 3)
- C, C++, and Java:
  - Use short-circuit evaluation for Boolean operations (&& and ||)
  - Also provide bitwise operators that are **not short circuit** (& and |)

Short-Circuit Evaluation

- Ada: programmers can specify either

Non-SC eval	SC eval
(x or y)	(x or else y)
(x and y)	(x and then y)

Control Structures

- Selection
- Iteration
  - Iterators
- Recursion
- Concurrency & non-determinism
  - Guarded commands

Iteration Based on Data Structures

- A data-based iteration statement uses a user-defined data structure and a user-defined function to go through the structure's elements
  - The function is called an **iterator**
  - The iterator is invoked at the beginning of each iteration
  - Each time it is invoked, an element from the data structure is returned
  - Elements are returned in a particular order

A Java Implementation for Iterator

```
class BinTree<T> implements Iterable<T> {
    BinTree<T> left;
    BinTree<T> right;
    T val;
    ...
    // other methods: insert, delete, lookup, ...

    public Iterator<T> iterator() {
        return new TreeIterator(this);
    }
    private class TreeIterator implements Iterator<T> {
        private Stack<BinTree<T>> s = new Stack<BinTree<T>>();
        TreeIterator(BinTree<T> n) {
            if (n.val != null) s.push(n);
        }
        public boolean hasNext() {
            return !s.empty();
        }
        public T next() {
            if (!hasNext()) throw new NoSuchElementException();
            BinTree<T> n = s.pop();
            if (n.right != null) s.push(n.right);
            if (n.left != null) s.push(n.left);
            return n.val;
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

Guarded Commands

- New and quite different forms of selection and loop structures were suggested by Dijkstra (1975)
- We cover guarded commands because they are the basis for two linguistic mechanisms developed later for concurrent programming in two languages: CSP and Ada

Motivations of Guarded Commands

- To support a program design methodology that ensures correctness during development rather than relying on verification or testing of completed programs afterwards
- Also useful for concurrency
- Increased clarity in reasoning

Guarded Commands

- Two guarded forms
  - Selection (guarded if)
  - Iteration (guarded do)

Guarded Selection

```
if <boolean> -> <statement>
[] <boolean> -> <statement>
     ...
[] <boolean> -> <statement>
fi
```

- Sementics
  - When this construct is reached
    - Evaluate all boolean expressions
    - If more than one is true, choose one nondeterministically
    - If none is true, it is a runtime error
- Idea: **Forces** one to consider **all possibilities**

An Example

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

- If i = 0 and j > i, the construct chooses nondeterministically between the first and the third assignment statements
- If i == j and i ≠ 0, none of the conditions is true and a runtime error occurs

Guarded Selection

- The construction can be an elegant way to state that the order of execution, in some cases, is irrelevant

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

  - E.g., if x == y, it does not matter which we assign to max
  - This is a form of abstraction provided by the nondeterministic semantics

Guarded Iteration

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

- Semantics:
  - For each iteration
    - Evaluate all boolean expressions
    - If more than one is true, choose one nondeterministically, and then start loop again
    - If none is true, exit the loop
- Idea: if the order of evaluation is not important, the program should not specify one

An Example

```
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
```

- Given four integer variables: q1, q2, q3, and q4, rearrange the values so that $q1 \le q2 \le q3 \le q4$
- Without guarded iteration, one solution is to put the values into an array, sort the array, and then assigns the value back to the four variables

An Example

- While the solution with guarded iteration is not difficult, it requires a good deal of code
- There is considerably increased complexity in the implementation of the guarded commands over their conventional deterministic counterparts