COSC252: Programming Languages:

Basic Semantics: Expressions and Side Effects

Jeremy Bolton, PhD

Asst Teaching Professor

GEORGETOWN UNIVERSITY

Outline

- 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, need to be familiar with the orders of operator and operand evaluation
  - Based on languages grammar rules
- Essence of imperative languages is dominant role of assignment statements

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages

- Generally speaking arithmetic expressions consist of operators, operands, parentheses, and function calls

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 has one operand
- A binary operator has two operands
- A ternary operator has three operands

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
-
- Typical precedence levels
  - parentheses
  - unary operators
  - ** or ^ (if the language supports it)
  - *, /
  - +, -

Arithmetic Expressions: Operator Associativity Rule

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated

- Typical associativity rules
  - *Left to right (**left associative**), except exponentiation, which is right to left (**right associative**)*

- Precedence and associativity rules can be “overridden” with parentheses, which is there general purpose in expressions

Interesting Design Decisions: Expressions 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

  - One result of this is that these operators can all be overriden by application programs
- Scheme (and Common Lisp)
  - All arithmetic and logic operations are by explicitly called subprograms
  - `a + b * c` is coded as `(+ a (* b c))`

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An 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*
  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: evaluate all operands and operators first
  4. The most interesting case is when an operand is a function call

Arithmetic Expressions: ***Potential 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; e.g., for a parameter change:

```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(&a);
```

In class -- Try This

Functional Side Effects

- Two possible solutions to the problem
  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
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations
     - 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 in the program 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, not, and referential transparency is violated

That is, all variable value changes must be explicit

Referential Transparency (continued)

- Advantage of referential transparency
  - Semantics of a program is much easier to understand if it has referential transparency
- Because they do not have variables, programs in pure functional languages are referentially transparent
  - Functions cannot have 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

- Use of an operator for more than one purpose is called *operator overloading*
- Some are common (e.g., + for **`int`** and **`float`**)
- Some are potential trouble (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability

Overloaded Operators (continued)

- 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

Type Conversions

- 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., `float` to `int`

- 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`

Type Conversions: Mixed Mode

- 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 F#, there are no coercions in expressions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - `C: (int)angle`
  - `F#: float(sum)`
  - `C++: static_cast<int>(price)`

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 (!=, ~=, <, >)

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators
- 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 problem (assuming `LIST` is `length - 1` long)

GEORGETOWN UNIVERSITY

Short Circuit Evaluation (continued)

- 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 `|`)

- All logic operators in Ruby, Perl, ML, F#, and Python are short-circuit evaluated

- The good, the bad, and the why why why would you ever do that …
  - Short-circuit evaluation exposes the potential problem of side effects in expressions
    e.g. `(a > b) || (b++ / 3)`
  - Can be useful if used appropriately
    e.g. `(x != 0) && (input[ind / x] == 'c')`

Assignment Statements

- The general syntax

  `<target_var> <assign_operator> <expression>`

- The assignment operator

  = Java, BASIC, the C-based languages

  := Ada

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
```

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

- Disadvantage: another kind of expression side effect

Semantics of Assignment Statement

- Example:

  x = 5 + f(10);

  1. Evaluate LHS expression
  2. Update symbol table entry for “x”

- Note: Some ambiguity may result if the language in implicitly typed, or there is no explicit syntax to differentiate declaration from definition.

EG

```
{  x = 5;
…
        {
            x = 10;
        }
}
```