CS2510
Modern Programming Languages

Topic 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 operator and operand order of evaluation
- Imperative languages: dominant role of assignment statements

Arithmetic Expressions

- Arithmetic expression was a motivation for the development of the first programming languages
- Arithmetic expressions:
  - Operators
  - Operands
  - Parentheses
  - Function calls

Arithmetic Expressions: Design Issues

- 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 the operand evaluation side effects?
  - Does the language allow user-defined operator overloading?
  - What type mixing is allowed 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
  - ** (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, except **, which is right to left
  - Sometimes unary operators associate right to left (e.g., in FORTRAN)
- APL is different:
  - all operators have equal precedence and all operators associate right to left
- Very important: precedence and associativity rules can be over-ridden with parentheses

Expressions in Ruby and Scheme

Ruby

- Arithmetic, relational and assignment operators, array indexing, shifts, and bit-wise logic operators, are implemented as methods
- Consequence: operators can all be overridden by application programs

Scheme (and Common LISP)

- Arithmetic and logic operations via explicitly called subprograms

`a + b * c` is represented as **`(+ a (* b c))`**

Arithmetic Expressions: Conditional Expressions

- C-based languages (e.g., C, C++):

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

- Evaluates as:

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

Arithmetic Expressions: 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. Parenthesised expressions: evaluate all operands and operators first

Interesting case: when an operand is a function call

Arithmetic Expressions: Potentials for Side Effects

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

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

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 *referential transparency* if
  - any two expressions with the same value can be interchanged anywhere in the program, without affecting the program

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

- If **`fun`** has no side effects, **`result1`** equals **`result2`**
- Otherwise **`result1`** and **`result2`** may differ, and referential transparency is violated

Referential Transparency (continued)

- 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 as they don't have variables
  - Functions do not have a state to 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 loss, even when the operators make sense

Type Conversions

- A *narrowing conversion* – converts a value to a type that does not have all values of the original type
  - Example: conversion of a **`float`** value to an **`int`** value
- A *widening conversion* – converts a value to a type that includes at least approximations of all values of the original type (and perhaps more)
  - Example: conversion of an **`int`** value to an **`float`** value

Type Conversions: Mixed Mode

- A *mixed-mode expression* 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 Ada, there are virtually no coercions in expressions
- In ML and F#, there are no coercions in expressions

Explicit Type Conversions

- Called *casting* in C-based languages
- Examples
  - C: **`(int)angle`**
  - F#: **`float(sum)`**
- Note: F#'s syntax 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)
- Often ignored by the run-time system

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types
  - Evaluate to some Boolean representation
  - Operator symbols vary among languages:

    “!=” “/=” “~=” “.NE.” “<>” “#”
- JavaScript and PHP: additional relational operators: “**===**”and “**!==**”
  - Similar to their cousins “**==**” and “**!=**” except that they do not coerce their operands
- Ruby uses “**==**” as equality relation operator with coercions and “**eql?**” without coercions

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
- C89 has no Boolean type
  - **`int`** type with 0 for false and nonzero for true
- One odd characteristic of C's expressions:
  - **`a < b < c`** is a legal expression
  - The result is not what you might expect:
  1. Left operator is evaluated, producing 0 or 1
  2. The evaluation result is then compared with the third operand (i.e., **c**)

Short Circuit Evaluation

- Result of expression determined without having to evaluate all operands/operators
- Example:

`(13 * a) * (b / 13 – 1)`

If **`a`** is zero, 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)

Short Circuit Evaluation (continued)

- C, C++, Java: short-circuit evaluation for Boolean operators (&& and ||)
  - Bitwise Boolean operators not short-circuit (& and |)
- Ruby, Perl, ML, F#, and Python: logic operators are short-circuit evaluated
- Ada: programmer controls short-circuit evaluation
  - Specified with **`and then`** and **`or else`**
- Short-circuit evaluation exposes potential problem of side effects in expressions: in Java

```
(a > b) || ((b++) / 3)
```

Assignment Statements

- General syntax

  <target_var> <assign_operator> <expression>

- Assignment operator
  - In Fortran, BASIC, C-based languages: "**=**"
  - In Ada: "**:=**"
- Symbol "**=**" sometimes overloaded – same as equality test
  - Hence C-based languages use "**==**" as equality

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

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

Which is equivalent to

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

Assignment Statements: Compound Assignment Operators

- A shorthand for a commonly needed form of assignment
- Introduced in ALGOL; adopted by C and C-based languages
- Example

```
a = a + b
```

  can be written as

```
a += b
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages:
  - Combine increment/decrement operations with assignment
- Examples

```
sum = ++count
```

(`count` incremented, then assigned to `sum`)

```
sum = count++
```

(`count` assigned to `sum`, then incremented

```
count++
```

(`count` incremented)

```
-count++
```

(`count` incremented then negated)

Assignment as an Expression

- In C-based languages, Perl, JavaScript, 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`) used as conditional value for the **`while`** statement

- Disadvantage: another kind of expression side effect

Multiple Assignments

- Perl, Ruby, and Lua: multiple-target multiple-source assignments

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

- The following is also legal: an interchange:

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

Assignment in Functional Languages

- Identifiers in functional languages: names of values
- ML: names are bound to values with **`val`**

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

  If another **`val`** for `fruit` follows, it is a new and different name
- F#: **`let`** is like ML's **`val`**, except **`let`** also creates a new scope

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode
- Fortran, C, Perl, C++: *any numeric type value* can be assigned to any numeric type variable
- Java, C#: only *widening assignment coercions* are done
- Ada: there is no assignment coercion

Summary

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