Principles of Programming Languages

Expression

Introduction

- Expressions are the fundamental means of specifying computations in a PL
  - While variables are the means for specifying storage
- Primary use of expressions: assignment
  - Main purpose: change the value of a variable
  - Essence of all imperative PLs: expressions change contents of variables (computations change states)
- To understand expression evaluation, need to know orders of operator and operand evaluation
  - Dictated by associativity and precedence rules

Arithmetic Expressions

- Arithmetic expressions consist of operators, operands, parentheses, and function calls
  - Unary, binary, ternary (e.g., _?_:_) operators
- Implementation involves:
  - Fetching operands, usually from memory
  - Executing arithmetic operations on the operands
- Design issues for arithmetic expressions
  - Operator precedence/associativity rules?
  - Order of operand evaluation and their side effects?
  - Operator overloading?
  - Type mixing in expressions?

Operator Precedence Rules

- Define the order in which “adjacent” operators of different precedence levels are evaluated
  - Based on the hierarchy of operator priorities
- Typical precedence levels
  - parentheses
  - unary operators
  - ** (exponentiation, if the language supports it)
  - *, /
  - +, -

Operator Associativity Rule

- 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
- Precedence and associativity rules can be overridden with parentheses

Conditional Expressions

- Conditional expressions by ternary operator **`?:`**
  - C-based languages (e.g., C, C++), e.g.,

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

  - Evaluates as if written like

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

Operand Evaluation Order

- How operands in expressions are "evaluated"?
  - Variables: fetch the value from memory
  - Constants: sometimes fetched from memory; sometimes in the machine language instruction
  - Parenthesized expressions: evaluate all inside operands and operators first
  - Operands on the two sides of an operator: evaluation order is usually irrelevant, except when the operand may cause side effects, e.g.,

```
b = a + foo(&a);
```

Side Effects in Expressions

- Functional side effects: a function changes a two-way parameter or a non-local variable
  - i.e., change the state "external" to the function
- Problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression:

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

Order in which operand is evaluated first will make difference

Functional Side Effects

- Functions in pure mathematics do not have side effects, i.e., $y = f(x)$
  - Input, x, determines output, y; no states
- Same with pure functional programming languages
- Side effects occur due to von Neumann arch. and associated imperative PL and computation model (state machines)
  - Memory/processor, variables/expressions, state/state change

Functional Side Effects

- Solution 1: define the language by disallowing functional side effects
  - No two-way parameters in functions
  - No non-local references in functions
  - Disadvantage: inflexibility of one-way parameters and lack of non-local references
- Solution 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

Overloaded Operators

```
int a,b;

float x,y;

…

b = a + 3;

y = x + 3.0;
```

- We wish to use the same operator '+' to operate on integers and floating-point numbers
  - Let compiler make proper translation, e.g., ADD vs FADD
  - How about '+' to operate on two array variables?

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 troublesome (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability
- C++/C# allow user-defined overloaded operator
  - Users can define nonsense operations
  - Readability may suffer, even when operators make sense, e.g., need to check operand types to know

Type Conversions

```
int a,b;
float x,y;
a = y;
x = b;
b = y + a;
```

- How should data be converted for assignment?
- What kinds of data format should compiler use during evaluation of the expressions?

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`
  - Not always safe
- 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`
  - Usually safe but may lose accuracy

Type Conversions: Mixed Mode

- A mixed-mode expression is one that has operands of different types
  - Need type conversion implicitly or explicitly
- Implicit type conversion by compiler: coercion
  - Disadvantage: 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 to minimize errors due to mixed-mode expressions

Type Conversions

- Explicit type conversion by programmer: casting in C-based languages, e.g.,
  - C: `(int) angle`
  - Ada: `Float (Sum)`
- Causes of errors in expressions
  - Inherent limitations of arithmetic, e.g., division by zero
  - Limitations of computer arithmetic, e.g. overflow
- Errors often ignored by the run-time system

Relational Expressions

- Expressions using relational operators and operands of various types; evaluate to Boolean
  - Relational operators: compare values of 2 operands
  - Operator symbols vary among languages (!=, /=, ~=, .NE., <>, #)

Boolean Expressions

- Expressions using Boolean operators and Boolean operands, and evaluate to Boolean
  - Boolean operands: Boolean variables, Boolean constants, relational expressions
  - Example operators:

FORTRAN 77	**FORTRAN 90**	**C**	**Ada**
`.AND.`	`and`	`&&`	`and`
`.OR.`	`or`	`||`	`or`
`.NOT.`	`not`	`!`	`not`

No Boolean Type in C

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

Precedence Operators in C

Highest postfix ++, --

unary +, -, prefix ++, --, !

*, /, %

binary +, -

<, >, <=, >=

=, !=

&&

Lowest ||

Short Circuit Evaluation

- An expression in which the result is determined w/o evaluating all operands and/or operators

```
(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 < listlen) && (LIST[index]
  != key)
      index = index + 1;
```

  - When **`index==listlen`**, **`LIST[index]`** causes an indexing problem (if LIST has **`listlen-1`** elements)

Short Circuit Evaluation

- 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 |)
- Ada: programmer can specify either (short-circuit is specified with **`and then`** and **`or else`**)
- Short-circuit evaluation exposes the potential problem of side effects in expressions
  e.g., **`(a > b) || (b++ / 3)`**

Assignment Statements

- The general syntax

  <target_var> <assign_operator> <expression>

- The assignment operator
  - = FORTRAN, BASIC, the C-based languages
  - := ALGOLs, Pascal, 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)

Conditional Targets

- Conditional targets (Perl)

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

  - Which is equivalent to

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

Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C
- Example:

```
a = a + b
```

is written as

```
a += b
```

Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
- Examples:
  - `sum = ++count` (count incremented, assigned to sum)
  - `sum = count++` (count assigned to sum and then incremented)
  - `count++` (count incremented)
  - `-count++` (count incremented then negated)

Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result and can be used as operands

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

  - `ch = getchar()` is carried out; result is used as a conditional value for the `while` statement
  - Has expression side effect: `a=b+(c=d/b)-1`
  - Multiple-target assignment: `sum = count = 0;`
  - Hard to tell: `if (x = y)` and `if (x == y)`
- Perl and Ruby support list assignments, e.g.,

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

Mixed-Mode Assignment

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