Chapter 7
Expression and Assignment Statements

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

Arithmetic Expressions

- Their evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls
Arithmetic Expressions

• Design issues for arithmetic expressions:
  1. What are the operator precedence rules?
  2. What are the operator associativity rules?
  3. What is the order of operand evaluation?
  4. Are there restrictions on operand evaluation side effects?
  5. Does the language allow user-defined operator overloading?
  6. What mode mixing is allowed in expressions?

Arithmetic Expressions

• A unary operator has one operand
• A binary operator has two operands
• A ternary operator has three operands

Arithmetic Expressions

• Def: The operator precedence rules for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated (“adjacent” means they are separated by at most one operand)
• Typical precedence levels
  1. parentheses
  2. unary operators
  3. ** (if the language supports it)
  4. *, /,
  5. +, -
Arithmetic Expressions

• Def: 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., FORTRAN)

• APL is different; all operators have equal precedence and all operators associate right to left.

• Precedence and associativity rules can be overridden with parentheses.

Arithmetic Expressions

• Operand evaluation order
  – The process:
    1. Variables: just fetch the value
    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. Function references: The case of most interest
      • Order of evaluation is crucial

Arithmetic Expressions

• Functional side effects - when a function changes a two-way parameter or a nonlocal variable.

• The 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;
b = a + fun(&a);
/* Assume that fun changes its parameter */
  – Same problem with global variables.
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 nonlocal references in functions
     – Advantage: it works!
     – Disadvantage: Programmers want the flexibility of two-way parameters (what about C?) and nonlocal references

Functional Side Effects

• Two Possible Solutions to the Problem (continued):
  2. Write the language definition to demand that operand evaluation order be fixed
     – Disadvantage: limits some compiler optimizations

Arithmetic Expressions

• Conditional Expressions allowed?
  – C, C++, and Java (?)
  
  e.g.
  
  average = (count == 0)? 0 : sum / count
Overloaded Operators

- **Def**: 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, e.g. unary-binary-)
  - Some loss of readability
  - Can be avoided by introduction of new symbols (e.g., Pascal’s `div`)

Overloaded Operators

- C++ and Ada allow user-defined overloaded operators
- Potential problems:
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense

Type Conversions

- **Def**: 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`
- **Def**: 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

• Def: A **mixed-mode expression** is one that has operands of different types
• Def: A **coercion** is an implicit type conversion
• The disadvantage of coercions:
  – They decrease 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

Type Conversions

• Explicit Type Conversions
• Often called casts
e.g.
Ada:
FLOAT(INDEX)--INDEX is INTEGER type

Java:
(int)speed /*speed is float type*/

Type Conversions

• Errors in Expressions
• Caused by:
  – Inherent limitations of arithmetic
e.g. division by zero
  – Limitations of computer arithmetic
e.g. overflow
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 (!=, /=, .NE., <>, #)

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Operators:
    - FORTRAN 77    FORTRAN 90     C      Ada
      .AND.      and         &&   and
      .OR.       or          ||   or
      .NOT.      not         !    not
      xor

Relational and Boolean Expressions

- C has no Boolean type—it uses int type with 0 for false and nonzero 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
Short Circuit Evaluation

- Don’t evaluate the full expression if no need
  - \(4 \times x \times Y(x) \times z\%m \Rightarrow \text{eval if } x=0\)
- Suppose Java did not use short-circuit evaluation

```java
Problem: table look-up
index = 1;
while (index <= length) &&
  (mytable[index] != value)
  index++;
Both the expressions (index<length) and (mytable[index] != value) evaluated => out of bound error
```

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)
- FORTRAN 77: short circuit, but any side-affected place must be set to undefined
- Short-circuit evaluation exposes the potential problem of side effects in expressions e.g. \((a \leq b) || (b++ / 3)\) => \(b\) is changed only if \(a>b\)

Assignment Statements

- The operator symbol:
  - FORTRAN, BASIC, PL/I, C, C++, Java
  - ALGOLs, Pascal, Ada
- = Can be bad if it is overloaded for the relational operator for equality e.g. (PL/I) \(A = B = C\)
- Note difference from C
  (does not =)
Assignment Statements

• More complicated assignments:
  1. Multiple targets (PL/I)
     \[ A, B = 10 \]
  2. Conditional targets (C, C++, and Java)
     \[ \text{if}\( \text{first}\approx\text{true} \)? \text{total} : \text{subtotal} = 0 \]
  3. Compound assignment operators (C, C++, and Java)
     \[ \text{sum} += \text{next}; \]

Assignment Statements

• More complicated assignments (continued):
  4. Unary assignment operators (C, C++, and Java)
     \[ a++; \]

C, C++, and Java treat = as an arithmetic binary
operator
e.g.
\[ a = b * (c = d * 2 + 1) + 1 \]

This is inherited from ALGOL 68

Assignment Statements

• Assignment as an Expression
  – In C, C++, and Java, the assignment statement
    produces a result
  – So, they can be used as operands in expressions
e.g.
  \[ \text{while} \left( ((\text{ch} = \text{getchar}()) != \text{EOF}) \right) \{ \ldots \} \]
  – Disadvantage
    • Another kind of expression side effect
Mixed-Mode Assignment

- In FORTRAN, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done.
- In Pascal, integers can be assigned to reals, but reals cannot be assigned to integers (the programmer must specify whether the conversion from real to integer is truncated or rounded).
- In Java, only widening assignment coercions are done.
- In Ada, there is no assignment coercion.

Chapter 8
Statement-Level Control Structures

- Introduction
- Selection Statements
- Iterative Statements
- Unconditional Branching
- Guarded Commands
- Conclusions

PQ2

1. Find a CFG to generate all strings of 0s and 1s that end with 0 (i.e., \text{regex} \{0, 1\}^*0).
2. What language does the following CFG recognize:
   \[ S \rightarrow a \ S \ c \ | \ B \]
   \[ B \rightarrow b \ B \ c \ | \ \epsilon \]
3. Write a CFG for compound statements in C/C++.
4. What language does the following CFG recognize:
   \[ S \rightarrow L \ | \ L \ A2 \ | \ A2 \rightarrow L \ | \ D \ | \ L \ A3 \ | \ D \ A3 \]
   \[ A3 \rightarrow L \ | \ D \ | \ L \ A4 \ | \ D \ A4 \ | \ A4 \rightarrow L \ | \ D \ | \ L \ A5 \ | \ D \ A5 \]
   \[ A5 \rightarrow L \ | \ D \ | \ L \ A6 \ | \ D \ A6 \ | \ A6 \rightarrow L \ | \ D \]
   \[ L \rightarrow A \ | \ B \ | \ \ldots \ | \ Z \ | \ D \rightarrow 0 \ | \ \ldots \ | \ 9 \]
Introduction

• Levels of Control Flow:
  1. Within expressions
  2. Among program units
  3. Among program statements

Introduction

• Evolution:
  – FORTRAN I control statements were based directly on IBM 704 hardware
  – Much research and argument in the 1960s about the issue
  – One important result: It was proven that all flowcharts can be coded with only two-way selection and pretest logical loops

Introduction

• Def: A control structure is a control statement and the statements whose execution it controls
• Overall Design Question:
  – What control statements should a language have, beyond selection and pretest logical loops?
Selection Statements

- A selection statement provides the means of choosing between two or more paths of execution.
- Two general categories:
  - Two-way selectors
  - Multiple-way selectors

Two-Way Selection Statements

- Design Issues:
  1. What is the form and type of the control expression?
  2. How are the then and else clauses specified?
  3. How should the meaning of nested selectors be specified?

Examples

- FORTRAN IF: IF(boolean_expr) statement
- Problem: can select only a single statement; to select more, a GOTO must be used, as in the following example.
Two-Way Selection Statements

• FORTRAN example:
  
  IF (.NOT. condition) GOTO 20

  ... 

  20 CONTINUE

• Negative logic is bad for readability
• This problem was solved in FORTRAN 77
• Most later languages allow compounds for the selectable segment of their single-way selectors

Two-Way Selection Statements

• Examples
  – ALGOL 60 if:
    if (boolean_expr)
    then statement (the then clause)
    else statement (the else clause)
  – The statements could be single or compound

Two-Way Selection Statements

• Nested Selectors
• e.g. (Java) if ...
  if ...
  ... 
  else ...
• Which if gets the else? (dangling else problem)
• Java's static semantics rule: else goes with the nearest if
Two-Way Selection Statements

- ALGOL 60’s solution - disallow direct nesting
  
  if ... then begin
    if ... then
      then ...
    else ...
  end

- Advantage: readability