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

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

• Arithmetic expressions consist of
  – operators
  – operands
  – parentheses
  – function calls
Design Issues for Arithmetic Expressions

• What form do expressions take (infix, prefix, postfix)
• 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
  unary – !

• A binary operator has two operands
  + − * / % < <= > >= == !=

• A ternary operator has three operands
  ?:
Operator Precedence Rules

• The *operator precedence rules* for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated.

• Typical order of precedence:
  - Parentheses
  - Unary operators
  - ** (if the language supports it)
  - * / %
  - + –
Operator Associativity Rules

- 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.
- Both precedence and associativity rules can be overridden with parentheses.
Ruby Expressions

- 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 overridden by application programs
Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:
    \[
    \text{average} = (\text{count} == 0)? 0 : \text{sum} / \text{count}
    \]

- Evaluates as if written like
  \[
  \text{if (count == 0)}
  \text{average} = 0
  \text{else}
  \text{average} = \text{sum} / \text{count}
  \]
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
Potentials 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);
```
Avoiding Functional Side Effects

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
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
User-Defined Overloaded Operators

• C++ and C# allow user-defined overloaded operators

• 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`
Coercion

- 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 in mixed-mode expressions are coerced using widening conversions
Explicit Type Conversions

- Called *casting* in C–based languages
- Examples
  - C: \((\text{int})\text{angle}\)
  - Ada: \(\text{Float} \ (\text{Sum})\)

Note that Ada’s syntax is similar to that of function calls
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
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 Expressions

- Relational Expressions
  - Use relational operators and operands of various types (<, >, ==, !=, etc.)
  - Evaluate to some Boolean representation
  - Operator symbols used vary somewhat among languages (!=, /=, ^=, .NE., <>, #)

- JavaScript and PHP have two additional relational operator, === and !==
  - Similar to their cousins, == and !=, except that they do not coerce their operands
Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators

<table>
<thead>
<tr>
<th>FORTRAN 77</th>
<th>FORTRAN 90</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>and</td>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>.OR.</td>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NOT.</td>
<td>not</td>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xor</td>
</tr>
</tbody>
</table>
No Boolean Type in C

- C89 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:
  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., $c$)
Short Circuit Evaluation

• Occurs when the result is determined without evaluating all of the operands and/or operators

• Example: \((13*a) \times (b/13-1)\)
  
  If \(a\) is zero, there is no need to evaluate \((b/13-1)\)
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 |)

- Short-circuit evaluation exposes the potential problem of side effects in expressions e.g. (a > b) || (b++ / 3)
Assignment Statements

• The general syntax
  \(<\text{target\_var}>\ \text{assign\_operator}>\ \text{expression}>\)

• The assignment operator
  \(=\) FORTRAN, BASIC, the C–based languages
  \(:=\) ALGOL, Pascal, Ada

• Poor readability when \(=\) is overloaded for the relational operator for equality
Conditional Targets

• Conditional targets (Perl)

\[ ($\text{flag} \ ? \ $\text{total} : \ $\text{subtotal}) = 0 \]

Which is equivalent to

```perl
if ($flag) {
    $total = 0
} else {
    $subtotal = 0
}
```
Compound 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 \(\text{ (count incremented then added to sum)}\)

  sum = count++ \(\text{ (count added to sum then incremented)}\)

  count++ \(\text{ (count incremented)}\)

  -count++ \(\text{ (returns } -\text{count and then increments original value)}\)
Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result and can be used as operands.
- An example:
  
  ```
  while ((ch = getchar()) != EOF) {...}
  ```

1. `ch = getchar()` is carried out;
2. the result assigned to `ch`
3. `ch` is used as a conditional value for the `while` statement
List Assignments

- Perl, Python and Ruby support list assignments

e.g.,

```perl
($first, $second, $third) = (20, 30, 40);
a, b = (3, 4)
```