Organization of Programming Languages CS3200 / 5200N

Lecture 07

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Control Flow

- **Control flow** = the flow of control, or execution sequence, in a program.

- Levels of control flow:
  1. Within **expressions**.
  2. Among **program statements**.
  3. Among **program units**.

Expressions

- Expressions are the fundamental means of specifying computations in a programming language:
  1. **Arithmetic** expressions.
  2. **Relational** expressions.
  3. **Boolean** expressions.

- The control flow in expression evaluation is determined by:
  1. The order of operator evaluation:
     - **Associativity**;
     - **Precedence**.
  2. The order of operand evaluation.

Arithmetic Expressions

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

- Arithmetic expressions consist of:
  - operators;
    - unary, binary.
  - operands;
  - parentheses;
  - function calls;

Arithmetic Expressions: Design Issues

- Operator precedence rules?
- Operator associativity rules?
- Operator overloading?

- Order of operand evaluation?
- Operand evaluation side effects?

- Type mixing in 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;
  - ** (where supported by the language);
  - *, /
  - +, –

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.

- Precedence and associativity rules can be overriden with parentheses:
  - When unsure, some programmers use parantheses ⇒ reduced readability.
  - Know thy language, its operators and their precedence rules!

Operator Overloading

- **Operator overloading =** the use of an operator for more than one purpose.
- Some are common (e.g., + for `int` and `float`).
- Some are potential trouble (e.g., `*`, `&` in C and C++):
  - Loss of readability.
  - Loss of compiler error detection:
    - omission of an operand should be a detectable error
  - Can be avoided by introduction of new symbols:
    - e.g., Pascal's **`div`** for integer division.

Operator Overloading

- C++, Ada, Fortran 95, and C# allow user-defined overloaded operators.
  - Problem: users can define nonsense operations.
- In Ruby, all arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bit-wise logic operators, are implemented as methods:
  - These operators can all be overriden by application programs.

Operands Evaluation & 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. Function calls:
   – potential for *side effects* ⇒ **operand evaluation order** is relevant.

Functional 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:

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

Functional Side Effects: Possible Solutions

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

- **Referential Transparency**: an expression can be substituted with its value, without changing the effects of the program.
  - Functional side effects violate referential transparency.
- Advantages of referential transparency:
  - Program semantics is much easier to understand.
- Programs written in functional programming languages are referential transparent:
  - no variables $\Rightarrow$ functions cannot have state.
  - value of function depends only on its parameters and global constants.

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`.

- Implicit type conversions i.e. **coercions**.
- Explicit type conversions i.e. **casts** in C/C++/Java:
  - C: `(int)angle`
  - Ada: `Float (Sum)`

Mixed-Mode Expressions

- A **mixed-mode expression** is one that has operators with operands of different types.
  - Type coercions are used in mixed-mode expressions to convert all operands to the same type.
- Disadvantage of coercions:
  - They decrease the type error detection ability of the compiler.
- Scenarios:
  - All numeric types are coerced in expressions, using widening conversions (most languages).
  - In Ada, there are virtually no coercions in expressions.

Relational Expressions

- Relational Expressions
  - Use relational operators and operands of various types.
  - Evaluate to some Boolean representation.
  - Always lower precedence than the arithmetic operators.
  - 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.
  - Ex: "7" == 7 vs. "7" === 7.

Boolean Expressions

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

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

Boolean Expressions in C/C++

- C versions prior to C99 have no Boolean type:
  - use `int` type with 0 for false and nonzero for true.
- Odd characteristic of C/C++ boolean expressions:
  - arithmetic expressions can be used for Boolean 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.
- Disadvantages:
  - loss in readability.
  - loss in type error detection.

Short-Circuit Evaluation

- The result of an expressions 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` has `length` elements).

Short-Circuit Evaluation

- C, C++, and Java:
  - use short-circuit evaluation for the usual Boolean ops (`&&`, `||`).
  - provide bitwise Boolean operators that are not short circuit (`&`, `|`).

- Ada:
  - programmer can specify either:
    - short-circuit is specified with `and then` and `or else`.

- Short-circuit evaluation + side effects ⇒ subtle errors:
  - Example: `(a > b) || (b++ / 3)`

Simple Assignment Statements

- The general syntax:

```
<target_var> <assign_operator> <expression>
```

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

Assignments with Conditional Targets

- Conditional targets (C++, Perl):

```
flag ? total : subtotal = 0;
```

Equivalent to:

```
if (flag)
  total = 0;
else
  subtotal = 0;
```

Compound Assignment Operators

- A shorthand method of specifying a commonly needed form of assignment:

```
a = a <op> b
```

- Introduced in ALGOL 68, adopted by C based languages.
- Example:

```
a = a + b
```

is written as

```
a += b
```

Unary Assignment Operators

- Unary assignment operators combine increment and decrement operations with assignment.
- Perl, JavaScript, in C–based languages.
- Examples:

  `sum = ++count` (`count` incremented, `count` assigned to `sum`).

  `sum = count++` (`count` assigned to `sum`, `count` incremented).

  `count++` (`count` incremented)

Assignments as Expressions

- Perl, JavaScript, and C–based: the assignment statement produces a result that can be used as an expression.
- Examples:
  - `while ((ch = getchar())!= EOF){...}`
    - `ch = getchar()` is carried out; the result (assigned to `ch`) is used as a conditional value for the `while` statement.
  - `a = b = 0`
- Problems:
  - loss of error detection: `if (x=y)` instead of `if (x == y)`

List Assignments

- List assignment: multiple source, multiple target.

- Perl, Python, Ruby support list assignments:

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

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

Mixed-Mode Assignments

- Assignment statements can also be mixed-mode, for example:

```
int a, b;
float c;
c = a / b;
```

- In Fortran, C, and C++, any numeric type value can be assigned to any numeric type variable.
- In Java, only widening assignment coercions are allowed.
- In Ada, there is no assignment coercion.

Reading Assignment

Chapter 7.