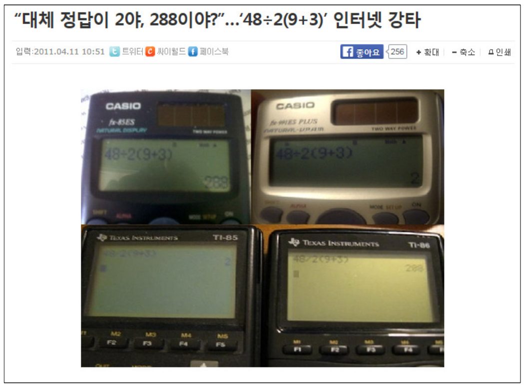
Chapter 7

Expressions and the Assignment Statement



http://news.kukinews.com/article/view.asp?page=1&gCode=cul&arcid=0004842193&code=41171111&cp=nv1

*“The operator evaluation order of expressions is governed by the **associativity** and **precedence rules** of the language. In the environment of von-Neumann architecture, **assignment** is the most fundamental statement.”*

7.2 Arithmetic Expressions

- ***Automatic evaluation of arithmetic expressions*** similar to those found in mathematics was **one of the primary goals of the first programming languages**

(1) Operator Evaluation Order

- Hierarchy of evaluation priorities

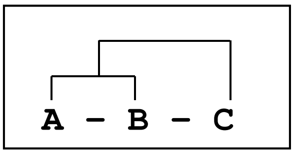
- ***Operator Precedence rules***
  - **the order** in which “**adjacent**” operators of different precedence levels are evaluated
    - ⇔ `A + B * C`
  - **identity operator** (unary operator : no effect on its operand)
    - ⇔ `+A, A + (-B) *C`
  - the operator precedence rules of the common imperative languages are **nearly all the same**
  - Precedence of arithmetic operators
    - ⇔ FORTRAN : `**` -> `*`,`/` -> all `+`,`-`
    - ⇔ Pascal : `*`,`/`,`div`,`mod` -> all `+`,`-`
    - ⇔ ANSI C : `++`,`--`,`unary +`,`-` -> `*`,`/`,`%` -> `binary +`,`-`

highest ← → *lowest*

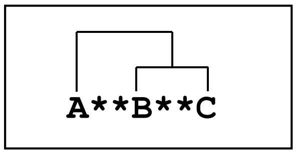
A **unary** operator has one operand
A **binary** operator has two operands
A **ternary** operator has three operands

- ***Associativity Rule***
  - When expression contains two adjacent occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the associativity rules of the language

    `A – B + C – D`
  - Associativity rules
    - ⇔ FORTRAN
      - ⇒ left : `*,/,+,–` right : `**`
    - ⇔ Pascal
      - ⇒ left : `all`
    - ⇔ ANSI C
      - ⇒ left : `*,/,%,binary +, binary –`
      - ⇒ right : `++, --, unary +, unary –`
    - ⇔ APL
      - ⇒ No precedence, only associative rule (Right -> Left)



left-> right



right -> left

  - If the compiler is allowed to reorder the evaluation of operators, it may be able to produce slightly faster code for expression evaluation (as in C) (because the arithmetic operations are mathematically associative)
    - ⇔ A + B + C + D
    - ⇔ Overflow ? (Integer addition on a computer is not associative !)
- ***Parentheses***
  - Programmers can alter the precedence and associativity rules by placing parentheses in expression
    - ⇔ `(A + B) * C`

(2) Operand Evaluation Order

- If operands of an operator have side effect, then operand evaluation order is important

- ***Side Effect***
  - A side effect of a function, called *a functional side effect*, occurs when the function either changes one of it parameters or a global variable

```
procedure sub1(....) ;
    var a : integer
    function fun(x:integer) : integer ;
        ....
        a := 27 ;
        return(5);
    end
    procedure sub2(....) ;
        ....
        a := 10 ;
        b := a + fun(b) ;
        print(b) ;
    end
end
```

an operand is a function call

  - How to handle it ?
    - ⇔ By disallowing functional side effect
    - ⇔ By language definitions (particular order)
      - ⇒ Java requires that operands appear to be evaluated in left-to-right order

(3) Conditional Expressions

- Sometimes `if-then-else` statements are used to perform a conditional expression assignment
  - In Pascal,

```
if (count = 0) then average := 0 ;
               else average := sum/count ;
```

  - In C (using ternary operator '?')

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

7.3 Overloaded Operator

- The multiple use of an operator is called ***operator overloading*** and generally thought to be acceptable, as long as readability and/or reliability do not suffer
  - Examples :
    - ⇔ '+' operator in FORTRAN
      - ⇒ `1 + 3, 1.0 + 3.0`
      - ⇒ `AVG = SUM / COUNT`

```
float  f ;
int i = 3, j = 2;
f = i/j ; /* f: ?? */
```

    - ⇔ '&' operator in C
      - ⇒ c = a & b ; c = &b ;
    - ⇔ '/' (*floating point division*) and '`div`' (*integer division*) operators in Pascal
    - ⇔ *user-defined overloaded operator* in Ada, C++, C#, F#
      - ⇒ The Ada compiler will choose the correct meaning when an overloaded operator is specified, based on the type of operands
      - ⇒ `A * B + C * D` (A,B,C,D : Matrix data type (2D Array))

a new operator defined by programmer

7.4 Type Conversions

- Languages that do allow ***mixed-mode expressions*** must define conventions, called ***coercions***, because computers usually do not have operations that use the operands of different types
  - ***coercion*** : an ***implicit type conversion*** that is initiated by compiler
  - ***casting*** : an ***explicit type conversions*** explicitly requested by programmers
- Type conversion
  - ***Narrowing conversion*** : converting an object to a type that cannot include all of the values of the original type (예 : converting a ***double*** to ***real)***
  - ***Widening conversion*** : converting an object to a type that can include at least approximations of all of the values of the original types (예 : converting a ***real*** to ***double)***
    - ⇔ always safe, but how about to convert *integer* to *float* ? Is it OK always ?
      - ⇒ some accuracy may be lost
        - → Integer 32 bit : 9 decimal digit, float 32 bit : 7 decimal digit
- Coercion design choices
  - In FORTRAN 77 :
    - ⇔ numeric data types : integer, real, double, complex
    - ⇔ all coercions are widening conversion
  - In (original) C
    - ⇔ numeric data types : int, short int, long int, float, double
    - ⇔ although *float* and *short int* are legitimate data types, they are always coerced to *double* and *int*, respectively, when they are appear in an expression or actual parameter list

- Mixed Mode Expression vs. Type Checking
  - Potential problems in coercions
    - ⇔ In FORTRAN 77,

```
INTEGER  A,B,C
REAL D
....
C = FUN(A+D)
```

```
function FUN (K:INTEGER){
….
}
```

no type checking in parameter passing in FORTRAN77

  - In Ada and Modula-2,
    - ⇔ do not allow mixing of integer and floating-point operands in expressions
- Explicit conversions (Casting)
  - In Ada and Modula-2,
    - ⇔ using the syntax of function call

```
AVG := FLOAT(SUM) / FLOAT(COUNT) ;
```

  - In C,

```
AVG = (int) SUM ;
```

- Errors in Expressions
  - Raise a Exception
    - ⇔ Overflow
    - ⇔ Underflow
    - ⇔ Divide by Zero

7.5 Relational and Boolean Expressions

- Relational Expression `(a > b)`
  - it has two operands and one relational operator
  - a relational operator is an operator that compares the values of its two operands
  - the value of relational expression is Boolean
  - the relational operators are usually overloaded for a variety of types
    - ⇔ the operation that determines the truth or falsehood of a relational expression depends on the operand types
  - relational operators always have lower precedence than the arithmetic operators; `z + 1 > 2 * b` → `(z + 1) > (2 * b)`
  - Syntax of relational operators

Operation	Pascal	Ada	C	FORTRAN 77
equal	=	=	==	.EQ.
not equal	<>	/=	!=	.NE.
greater than	>	>	>	.GT.
less than	<	<	<	.LT.
greater than or equal	>=	>=	>=	.GE.
less than or equal	<=	<=	<=	.LE.

string is corced

("7" == 7) : T
("7" === 7) : F

JavaScript, PHP
'===', '!=='
➔ do not coerce their operands

- **Boolean Expression**
  - it consists of boolean variables, boolean constants (`TRUE`, `FALSE`), relational expressions, boolean operators
    - ⇔ Boolean operators : `AND`, `OR`, `NOT`
      - ⇒ It also has precedence order
      - ⇒ In FORTRAN 77,

Highest :	`**`
	`*`, `/`
	`+`, `-`
	`//` (string catenation)
	`.EQ.`, `.NE.`, `.GT.`, `.LT.`, `.LE.`, `.GE.`
	`.NOT.`
	`.AND.`
	`.OR.`
Lowest	`.EQV.`, `.NEQV.`

evaluation order ?

```
A + B .GT. 2 * C .AND. K .NE. 0
```

  - In C,
    - ⇔ no Boolean types and thus no Boolean values
    - ⇔ numeric values are used to represent Boolean values
      - ⇒ zero : false
      - ⇒ all nonzero values : true
    - ⇔ *hard to detect errors in boolean expressions !*

in C,

```
b = 1 ;
if (a=b) {….}
```

```
a=11; b=22; c=3;
if (a<b<c) {….}
```

left associative

7.6 Short-Circuit Evaluation

- A ***short-circuit evaluation*** of an expression is one in which the result is determined ***without evaluating all of the operands and/or operators***

```
(13 * A) * (B / 13 -1)
```

if (A == 0)

```
(A >= 0) and (B < 10)
```

if (A >= 0) is False

- In Pascal,
  - most Pascal implementations do not use short-circuit evaluation
    -> sometimes causes some run-time errors

```
int list[10] ;
int listlen = 10 ;
index := 1 ;
while (index <= listlen) and (list[index] <> key) do
     index := index + 1 ;
```

- In FORTRAN,
  - implementor may choose not to evaluate any more of an expression than is necessary to determine the result
  - how to handle if unevaluated expression has a side effect ?
- In 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 `|`)

- In Ada,
  - allows the programmer to specify short-circuit evaluation of the Boolean operators AND and OR by using the two-word operators *and_then* and *or_else*

```
INDEX := 1 ;
while (INDEX <= LISTLEN) and_then (LIST[INDEX] \= key)
     loop
         INDEX := INDEX + 1 ;
     end loop ;
```

- In C and Modula-2
  - every evaluation of AND and OR expression is *short-circuit*
  - trade-off between *efficiency* and *responsibility*

7.7 The Assignment Statement

- one of the central constructs in imperative language
- provides a mechanism by which the user can dynamically change the binding of value to variable

- The Simple Assignment
  - Basic Form

```
<target_variable> <assignment_operator> <expression>
```

⇔ In FORTRAN, BASIC, PL/1, C,

⇒ use equal sign ('=') for the assignment operator

⇒ confused with relational operator

→ `A = B = C` (in PL/1)

⇔ In Algol 60,

⇒ use ':=' for the assignment operator

- Multiple Targets
  - allowing assignment of the expression value to more than one location

⇔ In PL/1

⇒ `SUM, TOTAL = 0`

- **Conditional Targets**
  - In C++

```
flag ? count1 : count2 = 0 ;
```
결과 : *l-value*

```
sum = flag ? cont1:count2 ;
```
결과 : *r-value*

- **Compound Assignment Operators**
  - It is a shorthand method for the assignment in which destination variable also appears as the first operand in the expression on the right side
    - ⇔ In C, (+=, −=, *=, /=)

```
sum = sum + value ;
⇒ sum += value ;
```

- **Unary Assignment Operators**
  - In C,
    - ⇔ "++" : for increment
    - ⇔ "--" : decrement
    - ⇔ as *prefix operators* : they precede the operands

```
sum = ++count ;   →   count = count+1;
                      sum = count ;
```

    - ⇔ as *postfix* operators : they follow their operands

```
sum = count++ ;   →   sum = count ;
                      count = count+1;
```

    - ⇔ as unary increment operator

```
count++ ;   →   count = count+1;
```


- **Assignment Statements as Operands**
  - In C, *the assignment statement produces a result*, which is the value assigned to the target. It can therefore be used as an operand in expressions

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

≠

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

  - it can lead to expressions that are very difficult to read and understand
  - allows the effect of *multiple-target assignments*

```
sum = count = 0 ;
```

  - a loss of error detection

```
if (x = y) ..... /* not a syntax error */
```

7.8 Mixed-Mode Assignment

- **Design question**
  - Does the type of expression have to be the same as the type of the variable being assigned, or can coercion be used in some case of type mismatch ?
  - In FORTRAN,

    ⇔ *the same coercion rules* for mixed type assignment that it uses for mixed type expressions; that is, many of possible type mixes are legal, with coercion freely applied
  - In Pascal,

    ⇔ includes some assignment coercion

    ⇒ integers can be assigned to floating-point variables
  - In Ada and Modula-2

    ⇔ *do not allow the coercion* of integer to floating-point in their assignment

```
int i ;
float a, b ;
i = a * i ;
```


- **Homework**

1. Assume the following rules of associativity and precedence for expressions

Precedence Highest	*, /, **not**
	+, –, &, **mod**
	– (unary)
	=, /=, < , <=, >=, >
	and
Lowest	**or**, **xor**
Associativity Left to right	

Show the order of evaluation of the following expressions by parenthesizing all subexpressions and placing a superscript on the right parenthesis to indicate order. For example, for the expression

`a + b * c + d` ⇨ $((a + (b * c)^1)^2 + d)^3$

① `a * b - 1 + c`
② `a * (b - 1) / c mod d`
③ `(a - b) / c & (d * e / a - 3)`
④ `-a or c = d and e`
⑤ `a > b xor c or d <= 17`
⑥ `-a + b`

2. Show the order of evaluation of the expressions of Problem 1, assuming that there are no precedence rules and all operators associate right to left.

3. Let the function **fun** and its usage be defined as

```
int fun(int *k) {
  *k += 4;
    return 3 * (*k) - 1;
}
```

```
void main() {
        int i = 10, j = 10, sum1, sum2;
        sum1 = (i / 2) + fun(&i);
        sum2 = fun(&j) + (j / 2);
}
```

What are the values of sum1 and sum2

① if the operands in the expressions are evaluated left to right ?

② if the operands in the expressions are evaluated right to left ?

4. Consider the following C program:

```
int fun(int *i) {
        *i += 5;
        return 4;
}
void main() {
        int x = 3;
        x = x + fun(&x);
}
```

What is the value of x after the assignment statement in main, assuming

① operands are evaluated left to right.

② operands are evaluated right to left.

5. Let the function **fun** and its usage be defined as

```
int a, b;
main() {
        a = 10;
        b = a + fun();
        printf("With the function call on the right, ");
        printf(" b is: %d\n", b);
        a = 10;
        b = fun() + a;
        printf("With the function call on the left, ");
        printf(" b is: %d\n", b);
}

fun() {
        a = a + 10;
}
```

Explain the results.