Lecture 05: Expressions

Expressions are the building blocks for arithmetic and Boolean expressions, as well as assignment statements. All programming languages therefore need expressions. This lecture discusses the topic under the following subheadings:

- Introduction
- Arithmetic Expressions
- Operator Overloading
- Type Conversions
- Boolean expressions
- Assignment statements

5.1 Introduction

In all programming languages, but particularly imperative (procedural) and OO languages, expressions and assignment statements are very important. Referring loosely to the grammar jargon of lecture 3, we may consider an expression as the combination of non-terminal and/or terminal symbols of a given grammar to make sense for the particular language impacted by that grammar. Moreover, recall that in lecture 3, we also looked at production rules for a grammar. To clarify, the production rules translate to the syntax rules for the language. Each language provides syntax for how programming constructions such as statements and expressions are formulated.

An expression typically consists of a combination of variables, optional subprogram call(s), and operators of the language in question.

Expressions are influenced by operators, operator precedence, type mismatches, data coercion, and short circuit evaluations. There are two types of expressions: arithmetic expressions and Boolean expressions. We shall examine both.

An assignment statement is used to assign a value to a variable. We shall also examine how this is done in different languages.

5.2 Arithmetic Expressions

Arithmetic expressions are used for evaluating values that will be used in assignment statements or other Boolean expressions. These expressions are based on the arithmetic operators, operator precedence, data coercion, and shortcut evaluations. Figure 5.1 illustrates the definition of an arithmetic expression (BNF notation used) in Java as well as C++.

5.2.1 Operators and Operator Precedence

Most imperative and OO languages inherit and support the basic binary operators from linear algebra (+, -, *, /, etc.). In most languages, these operators are used in the infix notation.

C-based languages also support unary operators such as ++ and --.

Operator precedence rules are similar but not identical in different languages. Figure 5.2 summarizes the precedence rules in Pascal, C-based language, and Ada. You will observe that C-based languages have more operators compared to Pascal or Ada.

Figure 5.1: Arithmetic Expression in C-based Languages

ArithExpression ::= <Literal> | <Variable> | <ShortcutExpression> |
<IncDecOpr> <Variable> |
<Variable> <IncDecOpr> |
[<ArithExpression> <ArithOperator> <ArithExpression>]

ShortcutExpression ::= <Variable> <Operator> = <Expression>

ArithOperator ::= + | - | * | / | % | = **IncDecOper** ::= ++ | --

Meaning of the Basic Arithmetic Operations

+	Addition
-	Subtraction or negation
*	Multiplication
/	Division
%	Modulus (remainder)
++	Increment by 1 (may be prefix or postfix) e.g. Counter++ (increment after use) ++Counter (increment before use)
--	Decrement by 1 (may be prefix or postfix)
=	Assignment. Thus Counter = Counter+1 is equivalent to ++ Counter

Clarification on Shortcut Expression:

<Variable> = <Variable> <Operator> <Expression> // may be shortened to
<Variable> <Operator> = <Expression>

```
// Examples
x = x + y;      /* is equivalent to */   x += y;
x = x – y;      /* is equivalent to */   x -= y;
x = x * y;      /* is equivalent to */   x *= y;
x = x % y;      /* is equivalent to */   x %= y;
x = x / y;      /* is equivalent to */   x /= y;
```

Figure 5.2: Operator Precedence Rules Comparison

Pascal:

```
( )
Unary -
NOT
* / div mod AND
+ - OR
= <> < <= > >= IN
```

C-based Languages:

```
( ) [ ] → .
! ~ ++ -- + - * & (<type>)
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= != <<= >>=
```

Ada:

```
( )
** abs
* / mod rem
Unary + -
Binary + -
```

5.2.2 Associatively

In most languages, the operators * and + are associative. The rule for associatively is inherited directly from linear algebra, as illustrated in the following examples:

- A * B * C = A * C * B = B * C * A = B * A * C = C * B * A = C * A * B
- A + B + C = A + C + B = B + C + A = B + A + C = C + B + A = C + A + B

5.2.3 Conditional Operator

C-based languages have a ternary conditional operator (?:) which implements a simple case if the if-then-else logic. The syntax for usage is as follows:

```
<Condition> ?: <Expression1> : <Expression2>;
```

This is equivalent to:

```
if <Condition>
   <Expression1>;
else <Expression2>;
```

5.3 Operator Overloading

Many languages allow operator overloading, some to a greater degree than others. Operator overloading is the facility to redefine the meaning of an operator, when used with respect to a context that is somewhat different from its original intent. In other words, the operator can be used in multiple contexts. Examples of operator overloading include the following:

- The + operator is used to add data items belonging to different numeric data types. In Java and more recently, C++, it is also used for string concatenation.
- The * operator is used for multiplying numeric data items. In C and C++, it is also used with pointers.

Languages such as C++, Ada, Fortran 95, and C# allow the programmer to overload most of the operators by creating operator functions so that they relate to complex objects. In this area, C++ is very prominent, allowing almost all its operators to be candidates for operator overloading (excluding only **new**, **delete**, →, the indirection, and the comma). Indirectly, Java approaches this flexibility of C++ by allowing manipulation of all objects as instances of the **Object** class.

5.4 Type Conversion

Some languages allow data type conversions. Data type conversion may be implicit or explicit. Implicit (or widening) conversions involve promotion from a lower ranger to a higher range (e.g. integer to real number).

Example 1:

```
// Works for C, C++, or Java
int x; double z, y;
z = y * x; // x is promoted to a double
```

5.4 Type Conversion (continued)

Explicit type conversion (narrowing conversion) involves demotion from a higher range to a lower range, or conversion across apparently dissimilar type ranges (e.g. object to string). This requires use of a casting operator, which in C-based languages, is simply specifying the target data type in parentheses.

Example 2:

```
// Works for C, C++, or Java
int x; double z, y;
y = (int) z * x; // z is demoted to an integer
```

Explicit casting is risky, and must only be done when the programmer is absolutely certain of what he/she is doing. However, when used appropriately, it is a powerful tool.

5.5 Boolean Expressions

Boolean expressions are expressions that evaluate to true or false. They involve the use of relational operators. Relational operators facilitate comparison of expressions. While the syntax of Boolean expressions varies across different languages, the format depicted in figure 5.3 is quite common.
Figure 5.4 provides some examples of relational operators in Pascal, C-based languages, and Ada.

Figure 5.3: Commonly Used Format for Boolean Expression

```
BooleanExp ::=  <Comparison> | <NOTSymbol> <BooleanExp> | <Comparison> <ANDSymbol> <BooleanExp> |
                <Comparison> <ORSymbol> <BooleanExp> | (BooleanExp>)

<Comparison> ::=  <Variable> <Operator> <Variable> | <BooleanVariable> |
                  <Comparison> <Connector> <Comparison>

<Connector> ::= <ANDSymbol> | <ORSymbol> | <NOTSymbol>

<Operator> ::=  // a list of valid Boolean operators defined by the language
```

Figure 5.4: Relational Operators Comparison

Pascal:

=	The left side is EQUAL to the right side.
>	The left side is GREATER THAN the right side.
<	The left side is LESS THAN the right side.
<>	The left side is NOT EQUAL to the right side.
>=	The left side is GREATER THAN or EQUAL to the right side.
<=	The left side is LESS THAN or EQUAL to the right side.
AND	The AND operator
NOT	The NOT operator
OR	The OR operator

C-based Languages:

Relational operators.	
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
&&	boolean AND
||	boolean OR

Figure 5.4: Relational Operators Comparison (continued)

Ada:

=	The left side is EQUAL to the right side.
>	The left side is GREATER THAN the right side.
<	The left side is LESS THAN the right side.
/=	The left side is NOT EQUAL to the right side.
>=	The left side is GREATER THAN or EQUAL to the right side.
<=	The left side is LESS THAN or EQUAL to the right side.
AND	The AND operator
NOT	The NOT operator
OR	The OR operator
XOR	The exclusive OR operator

5.6 Assignment Statements

In most programming languages, the syntax for an assignment statement is as follows:

```
<Variable> <AssignmentOperator> <Expression>;
```

C-based languages use the equal symbol (=) as the assignment operator. ALGOL60, Pascal, and other Pascal-like languages use the colon followed by the equal symbol (:=) as the assignment symbol.

C-based languages also use the shortcut operator symbols such as +=, *=, /=, -=, %=.

One outlier is FoxPro, which uses any of three syntax definitions for its assignment statements:

```
<Variable> = <Expression> |
Store <Expression> To <Variable> {, <Variable>} |
Replace <Variable> With <Expression>
```

5.7 Summary and Concluding Remarks

Let us summarize what has been covered in this lecture:

- An expression is the combination of non-terminal and/or terminal symbols of a given grammar to make sense for the particular language impacted by that grammar. Each language provides syntax for how programming constructions such as statements and expressions are formulated.
- Arithmetic expressions are used for evaluating values that will be used in assignment statements or other Boolean expressions.
- In constructing expressions in any given language, it is imperative to learn and understand operator precedence rules of the language.
- A Boolean expression is an expression that evaluates to true or false. These expressions ten to be similar in appearance even across language barriers. However, the actual Boolean operators tend to vary.
- Operator overloading — the ability to change the original meaning of an operator so that its applicability is expanded from its original scenario of relevance to other scenario(s) — is supported in many programming languages but at varying degrees.

In the next lecture, we will examine control structures, another set of principles that enjoy widespread commonality across different languages.

5.8 Recommended Reading

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 7.