Expressions, Assignments, and Control Structures

Learning Objective

This objective of this session is to examine issues associated with the syntax and semantics of programming language expressions, assignments, and control structures.

Overview

In the last session, we took a fairly detailed look at data types. We examined how primitive data types (e.g., char, float, int, etc.) are represented internally in a computer. We then moved on to more complex data types, such as arrays, records, unions, and pointers.

This session will deal with *expressions, assignments,* and *control structures*. We'll look at the *semantic* issues with expressions, such as the order of operator evaluation, data-type conversions, etc. Next, we'll consider the primary issues with assignments, such as conditional and multiple targets. Finally, we'll look at statement level control structures, addressing why they are needed and the tradeoffs (in terms of readability, writability, reliability, and run-time efficiency) of various control-structure constructs.

Expressions

An *expression* consists of a sequence of constants, variables, parentheses, function (or subprogram) calls, and operators, for example:

```
X + 2.0 * Y -SQRT(X)
```

In this example, 2.0 is a constant, X and Y are variables, SQRT is a function call, and + and - are operators. Note the distinction between an expression and an assignment: in addition to an expression, an assignment also has a target variable and an assignment operator. The expression shown above is part of the assignment shown below:

```
Z := X + 2.0 * Y - SQRT(X);
```

We'll look at issues associated with assignments in the next session. Here we'll focus on expressions.

Arithmetic Expressions

Historically, arithmetic expressions were invented to facilitate mathematical computations on a computer. Many, if not most, arithmetic expressions on a computer mimic their mathematical form. Furthermore, expressions form the heart of imperative languages. As mentioned earlier, expressions consist of constants, variables, parentheses, subprogram and function calls, and operators. Both the *syntax* and *semantics* of expressions need to be understood. Some of the issues involved include the order of operand evaluation, type mismatches and coercion, and short-circuit evaluations.

Expressions are built from **operators**, which can be:

- ***Unary***: operators with a single operand, e.g., -X
- **Binary**: operators with two operands, e.g., X - Y
- **Ternary**: operators with three operands, e.g., the '?' operator

Binary operators are usually **in-fix**, which means that they appear between the operands, such as in X - Y, where the minus sign is an in-fix operator.

1) Operator precedence rules (the order for evaluating different operators),
2) Operator associativity rules (the order for evaluating operators with the same precedence),
3) The overall order of operand evaluations,
4) Possible side effects of operand evaluation,
5) Operator overloading, and
6) Allowable mode (data type) mixing.

Let's look at each of these.

Operator Precedence: The expression A + B * C could be interpreted as (A + B) * C or as A + (B * C), with different results. Imperative languages usually have exponentiation at the highest level, followed by multiplication and division (equally) at the next level, followed by addition and subtraction (equally) at the lowest level. Most imperative languages mimic the mathematical rule My Dear Aunt Sally, meaning that the order of precedence for arithmetic operations is multiply, divide, add, and subtract.

The unary plus and minus operators pose a special problem. Their meaning as an operator is as follows: +X does nothing (except possibly an implicit type conversion), and -X simply changes the sign of the variable X. The problem arises when a unary operator follows a binary + or - operator in an expression, e.g., X + - Y. Parentheses are needed to clear up the confusion for the compiler, e.g., X + (-Y), so that the unary operation is carried out first.

Operator Associativity

Associativity determines the order of evaluation when operators have the same level of precedence, e.g., A + B - C + D - E. Most imperative languages evaluate from left-to-right, i.e., they are ***left-associative***.

Exponentiation can be a problem (note, however, that only FORTRAN and Ada have the exponentiation operator **). Ada doe not have associative exponentiation, e.g., X**Y**Z is illegal, i.e., parentheses must be used.

It's important to realize that associativity can lead to different results due to the occurrence of overflow or underflow. For example, consider the expression A - B + C - D, where A, B and C are large numbers that approach the limits of their underlying data types. The evaluation may succeed if carried out in the order shown. However, if the compiler (which might be trying to optimize operations) evaluates the expression as A + C - B - D, the evaluation may fail due to overflow (when A and C are summed). Most imperative languages allow the use of parentheses to change precedence and associativity, e.g., (A - B) + (C - D), forcing evaluation in the specified order.

Earlier we mentioned **ternary** operators, i.e., operators with three operands. The C, C++, and Java? operator is an example: The expression

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

translates as

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

Order of Operation Evaluation and Side Effects

Side effects can result from the order of evaluating expressions. For example, a side effect can occur when a subprogram changes a parameter or a global (or otherwise visible) variable, such in the expression

```
A + FUNC(A)
```

If FUNC(A) changes the value of A, then the expression will evaluate differently depending on the order of the operands A and FUNC(A). Preventing side effects puts difficult demands on compilers, and therefore is not addressed in many programming languages.

Operator Overloading

Overloaded operators are operators that are used for more than one purpose. Some languages have overloaded operators built into the language, while others allow the programmer to create them. A few examples follow:

- Java uses the + operator for addition, and to concatenate strings
- C uses the & operator for "AND" in Boolean expressions and as the "address-of" operator with pointers, as in ptr = &X;
- **+** and **-** signs are used as either binary or unary operators
- overloading "/" for float and int (integer) division can cause a loss of accuracy for int divisions, so some languages such as Pascal use a separate division operator for int, e.g., **div**

Ada, C++, and FORTRAN 90 allow the programmer to define operator overloads. Therefore, it is possible (but highly unrecommended) to reverse the meaning of widely recognized operators such as + and -. Such overloads would clearly have an adverse impact on the readability of the code.

Type Mixing and Conversions

A ***type conversion*** means changing a value from one (data) type to another. Type conversions usually result in *widening* or *narrowing* the range of values that can be stored. As an example, converting a two-byte **int** to a **float** widens the range from +/- 32768 to +/- 10^{38}, while converting a two-byte **int** to a two-byte **short int** doesn't affect the width. Widening conversions are usually safe, but narrowing conversions may not be due to the possibility of overflow. Also, widening conversions may increase the range of a variable, but decrease its precision.

There are two kinds of type conversions: *implicit* (coercion), and *explicit.* Coercion is an implicit type conversion, occurring in expressions with *mixed modes*, e.g., combinations of **int** and **float** types: X = A + B, with X **double**, A **float**, and B **int**. Coercion tends to decrease *reliability* since strong type checking by the compiler is not enforced.

Explicit type conversions are done under the programmer's control, usually with conversion functions, e.g., AVE := FLOAT(SUM)/FLOAT(COUNT), where the programmer has used the FLOAT function to explicitly change integer types to floats. In the C programming language, explicit conversions are known as *casts*, are are indicated by putting the target type in parentheses: **(int**) SUM explicitly changes the float SUM to an integer.

Relational and Boolean Expression

Relational expressions are binary; typically, relational expressions are used to compare the values of two operands. The value of a relational expression is Boolean (**T** (true) or **F** (false)), when Boolean is an included type in the languages. Relational operators are usually overloaded, so that they can apply to either **int** or **float** data types. It's interesting to note that FORTRAN used alpha relational operators (e.g., .LT., .LE., .GT., .EQ., representing "less than," "less than or equal," "greater than," and "equal," respectively). Also, relational operators <u>always</u> have lower precedence than arithmetic operators

Some examples of relational expressions in FORTRAN, C, and Ada:

FORTRAN: IF(I .LE. J) GO TO 100
C: **if**(a >= b) then ...
Ada: 4 > 2 **and** 6 /= 13/2 has the Boolean value "false"
(note: the Ada operator /= means "not equal to")

Boolean operators operate on Boolean values, i.e., **T** or **F.** A "truth table" for the Boolean operators **and**, **or**, **xor** (exclusive or), and **not** looks like this:

A	B	A and B	A or B	A xor B	not A
F	F	F	F	F	T
F	T	F	T	T	T
T	F	F	T	T	F
T	T	T	T	F	F

Boolean expressions can be constructed from Boolean components, e.g., variables, constant relational expressions that evaluate to Booleans, etc. The precedence rules for Boolean operators is language dependent. Usually, the unary **not** operator has the highest precedence.

Considering the C programming language's power and richness, it's perhaps surprising that it has no Boolean type (C++ does, however; it's called **bool**). C uses the integer value **int** 1 to indicate "true," and **int** 0 for "false," and relational operators in C are <u>left</u> associative. The lack of Booleans in C makes it a <u>less</u> readable language.

Short-Circuit Evaluation

Sometimes, it's possible to determine the result of an expression without evaluating the entire expression; this is called a ***short-circuit evaluation***. As an example, the evaluation of the expression (3 * A) * (B/2 + 1) can stop after the first set of parentheses if A = 0, since the value of the second set of parentheses won't affect the final value.

Short-circuit evaluations are useful in that they can increase run-time efficiency, but such evaluations may lead to undesirable ***side effects***. In the example shown in the previous paragraph, if the program depends on the value in the second set of parentheses, the value would not have been calculated, resulting in an error.

Generally, the best programming language designs allow the programmer to determine when to allow short-circuit evaluation. Such a choice makes the programmer cognizant of the problem, provides the capability to maximize run-time efficiency, and maximizes programming flexibility.

Assignments

The *syntax* of assignment statements is straightforward:

<target_variable> <assignment_operator> <expression>

There are two common notations for the assignment operator: the mathematical equal (=) sign (used in FORTRAN, BASIC, PL/I, C, and C++), and the "colon-equal" sign (:=) that was pioneered in ALGOL 60 and now used in many other languages, including Ada and Pascal.

You may wonder about the "colon-equal" syntax used by some languages. There are at least two good reasons for using that syntax:

- accidental assignments may result: consider the C statement

  if(a=b) c;

  here, the intent is to check the variables a and b for equality. However, since the assignment operator in C is the equal sign, the variables a and b always end up equal, and what was meant to be the Boolean expression (a = = b) will appear to be true unless b is 0, since the assignment operator returns the assigned value as a side effect. The following C code illustrates this.

```
#include<stdio.h>
void main (void)
{
 int a=5,b;
 for (b = -3; b<=3; b++){
    printf("for b=%d, (a=b) provides %d and ", b,(a=b) );
    if (a=b)
      printf("(a=b) evaluated True\n");
    else
      printf("(a=b) evaluated False\n");
    }
}
```

```
/*This produced the following:
for b=-3, (a=b) provides -3 and (a=b) evaluated True
for b=-2, (a=b) provides -2 and (a=b) evaluated True
for b=-1, (a=b) provides -1 and (a=b) evaluated True
for b=0, (a=b) provides 0 and (a=b) evaluated False
for b=1, (a=b) provides 1 and (a=b) evaluated True
for b=2, (a=b) provides 2 and (a=b) evaluated True
for b=3, (a=b) provides 3 and (a=b) evaluated True
Press any key to continue*/
```

- Consider the FORTRAN statement X = X + 1 (i.e., increment the value of X by 1). While a programmer would be happy with this notation, a mathematician would cringe! The := notation means "assign the value of," not "is equal to."

Multiple target variables are a handy construct available in some languages; in PL/I: SUM, TOTAL = 0 (assigns the value 0 to both variables), and in C, C++, and Java: Sum = Total = 0; (does the same thing, using different syntax).

C++ and Java allow a *conditional target*, with the syntax:

flag **?** count1 : count2 = 0;

which is equivalent to (but perhaps not as readable as):

if(flag) count1 = 0; **else** count2 = 0;

Compound assignment operators allow short-hand expressions, such as:

sum += value; which is equivalent to sum = sum + value;

Java, C, and C++ all provide the useful *unary* prefix and postfix operators ++ and --, with the following syntax:

sum = ++ count; which means:

count = count + 1;
sum = count;

and the postfix version:

sum = count++; which means

sum = count;
count = count + 1;

The unary prefix and postfix operators can be used as assignments:
++count; and count++; are both equivalent to count = count + 1;

As was illustrated above for C, *assignments as expressions* are legal in C, C++, and Java, for example:

a = b +(c = d/b++) -1;

In this example, the assignment c = d/b++ would be evaluated first, and then used as an expression in the containing statement. Note that using assignments as expressions results in code that is not particularly readable or reliable, e.g., as we just saw above, the statement **if**(a = b); could be correct, or it might be a typo for **if**(a == b).

Finally, *mixed-mode assignments*, like mixed-mode expressions, are legal in many languages, but using different rules:

- FORTRAN, C, and C++ use coercion as in expressions
- Pascal allows **int** to be coerced into **float**, but not vice versa.
- Java allows assignment coercion only if it's widening.

The following is an example of a C mixed-mode assignment:

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

This statement will divide the two integers, and coerce the result into a float which is then assigned to the variable c.

Overview of Control Structures

Control statements are a necessary artifact of imperative languages. As you will recall, imperative languages focus on evaluating expressions and assigning values to variables. Unfortunately, for a program to be useful, that is not enough. Two additional mechanisms are needed to allow useful programs to be built:

- A way to control the path of statement execution
- A way to repeatedly execute *groups* of statements.

Control statements are designed to provide these capabilities.

Considerable research has gone into the investigation of what constitutes an adequate/useful set of control statements, in contrast to the early days of FORTRAN which just wrapped higher-level language around existing IBM 704 statements.

One of the best known, and most notorious, control statements is the **goto** (unconditional branch) statement. We'll see later in this session the power of the **goto** statement, and its dangers, and we'll see how the choice of control statements has a large impact on the readability and writability of programming languages. In addition, we'll see that control statements can be used to build *control structures*, and that most research has shown that control structures should have only one entry point and one exit point.

Compound Statements

Control-statement design is simplified if there is a construct to group statements together as a single **compound** statement. ALGOL 60 was the first to do so with its **begin-end** construct; an example is shown below:

```
begin
statement_1;
...
statement_n;
end
```

If we add data types to a compound statement, the result is called a **block** (with all the implications of scope).

Different programming languages have different capabilities and syntax for compound statements. Pascal allows compound statements (using **begin** and **end** reserved words), but does not allow **blocks**. C, C++, and Java use braces { } to designate compound statements and blocks, as exemplified by this piece of recursion code:

```
if((value >= 0)&&(value <= 34))
        {result = fact(value);
         printf("\n%! %e", value, result);
        }
```

The point of using a compound statement in the above example is to allow for the execution skipping all the statements between the braces, in the event the **if** statement argument is "false." Without the braces, the semantics of C, C++, and Java would only skip the execcution of the first statement after the **if** in the event the Boolean evaluated to "false."

It's good programming practice to allow only one entry into control structures (usually from the top). Multiple entries would require **goto** statements, which results in lowered readability and maintainability. On the other hand, multiple *exits* from control structures, including compound statements and blocks, are considered useful, and can enhance execution efficiency.

Selection Statements

Selection statements choose between two or more execution paths in a program, and may be categorized as two-way, or n-way (i.e., more than two-way). Design issues include

i) the form and type of expression controlling the selection,
ii) what is selected (e.g., a single statement, a sequence of statements, or a compound statement), and
iii) how selection statements can be nested.

An example of a two-way FORTRAN selection statement is:

```
IF(I .EQ. J) I = I + 1
```

A disadvantage of this construct is that if the Boolean expression (I .EQ. J) tests "true," only a single statement (I = I + 1 , in this case) can be executed. This limitation led to rampant use of

the **goto** statement. Other languages, such as ALGOL 60, overcome this deficiency through the use of compound statements.

ALGOL 60 also introduced two-way selectors, which have since worked their way into many other languages. The syntax is:

```
if(...) then
(then clause) - a statement or compound statement
else
(else clause) - a statement or compound statement
```

This construct has the advantage of being both writable and readable.

Selectors can be nested, but this can lead to confusion. The confusion can be minimized through the use of reserved words such as **end if**, to delineate the end of **if** clauses.

Sometimes, a programmer will want the code to select from more than two statements or compound statements, and will use an N-way selector. FORTRAN I included a three-way selector. A drawback of this construct is the heavy use of **goto** statements.

An example from Hastings book (please refer to Recommended Readings) of a modern (C, C++) N-way selector is:

```
switch (expression)
{
  case value_1:
     statements_1;
     break;
  case value_2:
     statements_2;
     break;
...
   case value_N;
      statements_N;
      break;
   default:
       statements_d;
}
```

In this example, the value of the arithmetic expression after the reserved word **switch** will determine which **case** statement to execute. Note that the syntax allows for multiple statements to be executed for each **case**. The **break** statement is a de facto **goto**, unconditionally branching control out of the N-selector construct after the selected code has executed. In the event that the expression in the **switch** statement doesn't match any of the **case** values, the **default** code is executed (it might contain an error message).

N-way selectors such as this example are now widely used in several modern languages. The construct has the advantage of writability, readability, reliability (e.g., because of the **default**

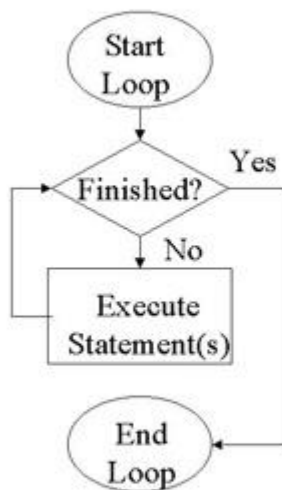
statement), and efficiency (e.g., because of the **break** statement). While the break statement increases efficiency, it also presents a reliability problem.

Take a moment to look back at the C source code for Binary (the lab exercise that was distributed in Session 5). Binary uses a 6-way selector (a *switch)* to determine which data type the user would like to convert. Note that the construct uses a *default* statement to handle inputs other than 1 to 6.

Iterative Statements

Iteration is the essence of imperative programming languages. Iteration, according to Webster, is *repetitious; repeating or repeated.* Usually, when used in the context of programming, iteration refers to the execution of a single statement (or more commonly, a group of statements), statement(s) that are executed over and over.

When a programmer wishes to iteratively execute a piece of code, two design issues arise: how to control the iteration, and where in the loop (e.g., at the beginning or at the end) is the iteration controlled. As shown here in the figure, the iteration is controlled at the beginning of the loop. It could be controlled by placing the test ("Finished") at the end of the loop instead.



The iteration is often controlled by creating a simple counter. Suppose we want to execute some statements ten times. The counter would be initialized to zero, and then incremented by one prior to (in this example) executing the statement. After executing the statements, the counter would be tested to determined if its value is less than ten; if not, the loop is finished. This type of loop is said to be ***counter controlled.***

Other means of testing for completion ("Finished?") are possible. If searching data records for the name 'John Smith,' the control statement could test "is current name equal to 'John Smith?' This type of loop is said to be ***logically controlled.***

Some languages provide ***user-located loop control mechanisms,*** so that the programmer can determine where and how the iteration loop is controlled.

Iteration statement(s), together with the body of the loop (the statements being executed in the above figure), are called the ***iteration construct.***

Now, let's look at the the three approaches to controlling loops in a bit more detail.

Counter-Controlled Loops

We'll start by dissecting a FORTRAN loop:

```
      INTEGER I
      DO 1 I = 1, 10
      SUM = SUM + 1
    1 CONTINUE
```

Note the syntax of the loop-control statement:

DO label variable = initial, terminal [,stepsize]

where the brackets indicate that the stepsize parameter is optional.

The integer variable I in this example is the ***loop variable***. The ***initial*** and ***final values*** of the loop variable are 1 and 10, respectively, and the ***stepsize*** is (by default in this example) 1, the default value in FORTRAN. A step size of 2 would have been indicated with the syntax DO 1 I = 1, 10, 2 which would have resulted in the loop variable taking the values 1, 3, 5, 7, and 9.

Versions of FORTRAN (versions I - IV) prior to FORTRAN 77 used ***posttest*** control, meaning that the value of the loop variable is checked after loop execution. This means that every loop is executed at least once, even if the initial value of the loop variable is greater than the final value. FORTRAN 77 introduced the more reasonable ***pretest*** control, which checks the loop variable against the final value prior to executing the loop.

An important semantic detail of FORTRAN **DO** loops is that they can be entered only from the top. This enhances the language's readability and reliability.

The important design issues for counter-controlled loops are

i) the type and scope of loop variables,
ii) the value of the loop variable at loop termination,
iii) the ability within a language to change a loop variable within the loop, and
iv) the issue of when and where the loop parameters are evaluated.

Binary includes a counter-controlled loop; take a look at its code and see if you can find it.

Logically Controlled Loops

Logically controlled loops are controlled by a Boolean expression rather than a counter. Many of the design issues relevant to counter-controlled loops carry over, such as whether to pre- or post-test.

C, C++, and Java implement logically controlled loops differently, depending on whether it's a pre-test or post-test loop. For a pre-test loop, the syntax is:

```
while (expression)
   {loop body}
```

and for a post-test loop:

```
do
   {loop body}
while (expression)
```

FORTRAN 77 and 90 do not have a pre-test or a post-test logical loop; Ada has a pre-test but not a post-test loop construct.

Binary has a pre-test logical loop. See if you can locate it, and understand how it works, in particular, the syntax/semantics of the loop-control expression.

User-Located Loop Control

Study the following example:

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
  }
```

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
  }
```

On encountering a negative, the **continue** command avoids the assignment by continuing with the next step in the iteration. The **break** command avoids the assignment by terminating the iteration. This construct obviously provides the programmer with enormous flexibility.

Unconditional Branching

An ***unconditional branch*** statement can be used to transfer execution control to a specified place in a program. Perhaps the most infamous example of an unconditional branch is the **goto** statement. At the risk of offending Shakespeare, to **goto** or not to **goto**, that is the eternal question in programming languages.

First, some arguments in favor of including a **goto** statement in a language:

- The **goto** statement is very powerful, allowing for control transfer upward or downward in a program,
- The **goto** allows for very flexible programs.

The arguments against the **goto** include:

- It has a high potential to make programs virtually unreadable,
- "The **goto** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program" - Dijkstra

The mess of one's program referred to by Dijkstra is often referred to as "spaghetti code," meaning that it is a pile of intertwined, complex statements coupled via **goto**s.

Despite these dangers, most imperative languages include some form of **goto**; Java and Modula-2 are notable exceptions.

The *syntax* of the **goto** usually includes some sort of label for the target statement. FORTRAN and Pascal use unsigned integer constants for their labels, as in this FORTRAN example:

```
      GO TO 10
       ...
   10 CONTINUE
```

Here, the unsigned integer label 10 is the target for the **goto** statement. Interestingly enough, PL/I allow labels to be variables, making the language's readability a potential nightmare.

Compromise approaches restricting the use of **goto** statements exist, such as that used by Pascal. In Pascal, labels must be declared, they have limited scope, and they must be constants. Furthermore, Pascal labels (**goto** targets) are only active if the compound statement in which they appear is currently active.

To summarize, the use of the **goto** statement is generally considered poor software-engineering practice, and the trend (such as in Java), is to not include it in new languages.

Wrap Up

In this session we've again covered a lot of programming language concepts. We began by examining the distinction between expressions and assignments; the importance of understanding the semantics of operator precedence and associativity for various programming languages; the usefulness of overloaded operators; how type conversions work; the syntax and usefulness of relational and Boolean expressions, and the syntax and semantics of assignments.

It's important to again remind you that it is not the intent of this course to learn details of particular programming languages, such as the syntax of C assignment statements. Rather, the objective is to learn and appreciate the concepts involved, such as the concept of operator overloading. Having acquired an understanding of these concepts, it will be much easier to learn and understand the strengths and weaknesses of particular languages.