7. Expressions and Assignment Statements

7.1. Introduction

- operator precedence
- operator associativity
- order of operator and operand evaluation
- type mismatch
- type coercion
- short-circuit evaluation
- imperative languages are dominated by assignment statements

7.2. Arithmetic Expressions

- unary, binary, and ternary operators meaning an operator can have one, two, or three operands respectively
- Design issues include
    - What are the operator precedence rules?
    - What are the operator associativity rules?
    - What is the order of operand evaluation?
    - Are there restrictions on operand evaluation side effects?
    - Does the language allow user-defined operator overloading?
    - What type mixing is allowed in expressions?

7.2.1. Operator Evaluation Order

7.2.1.1. Precedence

- Operators are executed in the order of their precedence. Operators having higher precedence are executed before operators having lower precedence. For example, consider:

  $$a + b * c$$

  Both the addition operator $+$ and the multiplication operator $*$ are binary operators. The addition operator wants to bind to the two nearest operands a and b. The multiplication operator likewise wants to bind to the two nearest operands b and c. Operand b is in contention. The question is: which operator binds to operand b since b can only be bound to one operator. Precedence determines the selection of operands. The operator with the highest precedence binds the operand in contention. The multiplication operator has the highest precedence and it executes first finding the product $b * c$. The value of operand a is added to the product.

	Ruby	*C-Based Languages*	*Ada*
Higher	**	postfix ++,--	**, **abs**
	unary +,-	prefix ++,--, unary+,-	*,/,**mod**,**rem**
	*,/,%	*,/,%	unary +,-
Lower	binary +,-	binary +,-	binary +,-

- Unary minus versus binary operators

Legal	*Usually Not Legal (legal in C++)*
a+(-*b*)**c*	*a*+-*b***c*

Expression	*Meaning*	*Language(s)*	*Discussion*
-*a*/*b*	(-*a*)/*b* or –(*a*/*b*)		Equivalent
-*a***b*	(-*a*)**b* or –(*a***b*)		Equivalent
-*a****b*	-(*a****b*)	Fortran, Ruby, Visual Basic, Ada	** has higher precedence than unary -

7.2.1.2. Associativity

- When two or more operators have the same precedence in an expression, associativity determines the order of evaluation. Consider the following example:

$$\boldsymbol{a - b + c - d}$$

Operands b and c are in contention. The subtraction operator and the addition operator are contending for operand b and operand c. Because addition and subtraction associate to the left in almost all cases, the difference $a - b$ is computed first, then c is added to the difference and, finally d is subtracted from the sum. The parenthesized expression below shows the order of operation.

$$(\big((\boldsymbol{a} - \boldsymbol{b}) + \boldsymbol{c}\big) - \boldsymbol{d})$$

Note how the parentheses are grouped on the left. Whenever operators associate to the left, an equivalent event occurs. Parentheses pile up on the left.

An analogous situation occurs when operators associate to the right. Parentheses pile up on the right. Consider the following example in C++:

$$\boldsymbol{a = b = c = d + 1}$$

Obviously, we cannot assign b to a because we do not have a value for b yet. Assignment operators associate to the right in C, C++, and Java. First the sum $\boldsymbol{d} + \boldsymbol{1}$ is evaluated and assigned to $\boldsymbol{c}$. Then $\boldsymbol{c}$ is assigned to $\boldsymbol{b}$. Finally, $\boldsymbol{b}$ is assigned to $\boldsymbol{a}$.

$$(\boldsymbol{a} = \Big(\boldsymbol{b} = \big(\boldsymbol{c} = (\boldsymbol{d} + \boldsymbol{1})\big)\Big))$$

Note how parentheses pile up on the right.

In many languages the exponentiation operator associates to the right.

$$a ** b ** c$$

is

$$(a ** (b ** c))$$

However, in Ada the exponentiation operator is non associative forcing the use of parentheses.

Legal in Ada	*Not Legal in Ada*
*a*****(***b****c***)**	*a****b****c*
(*a****b***)*****c*	

7.2.1.3. Parentheses

- Programmers can alter the precedence and associativity rules by placing parentheses in expressions. For example:

$$(\boldsymbol{a}+\boldsymbol{b}) * \boldsymbol{c}$$
$$(\boldsymbol{a}+\boldsymbol{b})+(\boldsymbol{c}+\boldsymbol{d})$$

In the foregoing example, if the sum of a, b, and c cause an overflow and d is negative to the degree that an overflow can be prevented; the parentheses shown above could be used to prevent an overflow.

7.2.1.4. Ruby Expressions

- All arithmetic, relational, assignment, array indexing, shifts, and bit-wise logic operators are implemented as methods.

7.2.1.5. Conditional Expressions

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

7.2.2. Operand Evaluation Order

- Only when the evaluation of operands causes side effects is the order of evaluation important (only when an operand is a function call).

7.2.2.1. Side Effects

- A **side effect** of a function occurs when the function changes either one of its parameters or a global variable.
- Detrimental side effect

```
int a=5;
int f(){a=17; return 3;}
int main()
{    a=a+f();
     return 0;
}
```

If the expression *a*+*f*() is evaluated from left to right it value is **8**. However, if the expression is evaluated from right to left its value is **20**.

- Beneficial side effect

```
void swap(int& m,int& w){int b=m;m=w;w=b;}
int main()
{     int a(1),b(2);
      swap(a,b);
      return 0;
}
```

7.2.2.2. Referential Transparency and Side Effects

- A program has the property of **referential transparency** if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program.

  ```
  //A fragment that has the property of referential transparency
  result1=(fun(a)+b)/(fun(a)-c);
  temp=fun(a);
  result2=(temp+b)/(temp-c);
  ```

  If the function *fun* has no side effects, *result1* and *result2* will be equal, because the expressions assigned to them are equivalent. However, suppose *fun* has the side effect of adding 1 to either *b* or *c*. Then *result1* would not be equal to *result2*.

- The semantics of a referentially transparent program are much easier to understand because functions become equivalent to mathematical functions.
- Pure functional programs are referentially transparent and have been used to prove attributes of programs.

7.3. Overloaded Operators

- The multiple use of an operator is called **operator overloading**.

  ```
  int a(1),b(2),c;
  c=a&b;                   //bitwise and

  int* p=&a;               //address operator
  ```

- Problems.
  - o Using the same symbol for two completely unrelated operations is detrimental to readability.
  - o The simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler, because it is interpreted as an address-of operator. Such an error may be difficult to diagnose.

7.4. Type Conversions

- Type conversions are either narrowing or widening.
  - o A **narrowing conversion** converts a value to a type that cannot store even approximations of all of the values of the original type.

    ```
    int a; double pi(3.14159); a=(int)pi;
    ```

  - o A **widening conversion** converts a value to a type that can include at least approximations of all of the values of the original type.

    ```
    int a(37); double b; b=(double)a;
    ```

7.4.1. Coercion in Expressions

- A **mixed-mode expression** is one where an operator can have operands of different types.

  double *pi*(**3.14159**); **double** *r*(**3.5**); **double** *d*=**2***pi***r*;

  Integer constant 2 and variable *pi* have different types. The integer constant is coerced to type double prior to multiplication.

- Ada does not permit mixed mode expressions and forces the programmer to employ explicit type conversions.

  A:*Integer*;
  B,*C*,*D*:*Float*;
  …
  C:=*B***A*; **-- not legal in Ada**

7.4.2. Explicit Type Conversion

- Most languages provide some capability for performing explicit conversions.
- In C-based languages, explicit type conversions are performed using **casts**.

  int *a*(**37**); **double** *b*; *b*=(**double**)*a*;

- In Ada, explicit type conversions take the form of function calls.

  Float(*Sum*)

7.4.3. Errors in Expressions

- An **overflow** occurs when the result of an arithmetic operation is too large to be stored in the designated memory cell.
- An **underflow** occurs when the result of an arithmetic operation is too small to be stored in the designated memory cell.
- An **exception** can be thrown (raised) by the runtime environment in some languages when an overflow or underflow occurs.

7.5. Relational and Boolean Expressions

7.5.1. Relational Expressions

- A **relational operator** is an operator that compares the values of its two operands.

Language	C-based	Ada	Lua	Fortran 95	Pascal
Inequality operator	**!=**	**/=**	**~=**	**.NE.**	**<>**

- Javascript and PHP must convert and compare equality and inequality operators.

```
"7"==7     //The string "7" is coerced to a numeric 7 and the equality
           //operator returns true


"7"===7    //The string "7" is not coerced to a numeric 7 and the equality
           //operator returns false
```

- Relational operators always have lower precedence so arithmetic operations are performed first.

  a+1>2**b*

7.5.2. Boolean Expressions

- Common Boolean operations include AND, OR, NOT, and EXCLUSIVE-OR.
- "In the mathematics of Boolean algebra, the OR and the AND operators must have equal precedence." However, Rosen in Discrete Mathematics and Its Applications, 6th Ed. assigns precedence in the table below

TABLE 8 Precedence of Logical Operators

Operator	*Precedence*
$\neg$	1
$\wedge$	2
$\vee$	3
$\rightarrow$	4
$\leftrightarrow$	5

- In C-based languages, precedence of operators is given in the table below.

Operator	***Precedence***
postfix ++, --	*Highest*
unary +, - , prefix ++, --, !	
***, /, %**	
binary +, -	
<, >, <=, >=	
=, !=	
&&	
||	*Lowest*

- In Pascal.

Operator	***Precedence***
not	*Highest*
***, /, div, mod, and**	
unary +, -	
binary +, -, or	
=,<>,<,<=,>,>=,in	*Lowest*

0<=*a* and *a*<=10 {Invalid in Pascal }
0<=*a* && *a*<=10 //valid in C-based languages}

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 or operators.

```
(a>=0)&&(b<10)      //If a<0 then the expression (b<10) need not be evaluated

                    //Mandatory short-circuit evaluation
while ((index<listlen)&&(list[index]!=key)) index=index+1;

(a>b)||((b++)/3)    //(a>b) or (b++/3!=0) – b++ need not be evaluated
```

- Short-circuit evaluation can be specified in Ada.

```
Index:=1;
while (Index<=Listlen) and then (List(Index)/=Key)
    loop
    Index:=Index+1;
    end loop;
```

- Ruby, Perl, and Python implement short-circuit evaluation on all logical operators.

7.7. Assignment Statements

7.7.1. Simple Assignments

- ALGOL 60 pioneered the use **:=** as the assignment operator to distinguish it from the equality operator.

7.7.2. Conditional Targets

- Perl.

  ($*flag*?$*count1*:$*count2*)=0
  is equivalent to
  if ($*flag*) { $*count1*=0; } else { $*count2*=0;}

7.7.3. Compound Assignment Operators

- A **compound assignment operator** is a shorthand method of specifying a commonly needed form of assignment.
- Introduced in Algol 68 and later modified slightly in C-based languages. Perl, JavaScript Python, and Ruby also support compound assignment operators.

```
sum+=value;     //sum=sum+value;
```

7.7.4. Unary Assignment Operators

- A **unary assignment operator** combine increment and decrement operations with assignment.

```
count++;        //count=count+1;
sum=++count;    //count=count+1; sum=count;
sum=count++;    //sum=count; count=count+1;
```

- When two unary operators apply to the same operand, the association is right to left.

```
-count++;       /- (count++) not (-count)++
```

7.7.5. Assignment as an Expression

- The value assigned is the value of the expression.

  while((*ch***=***getchar***())!=***EOF***){…}**

- Consider the statement.

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

  1. *c***=***d***/***b***;**
  2. *t***=***b***+***c***;**
  3. *a***=***t***-1;**

- Assignment expressions permit a multi-target assignment.

  *sum***=***count***=0;**

- Some operators are more equal than others. Consider

  if (*x***=***y***) …**
  or
  if (*x***==***y***) …**

7.7.6. List Assignment

- Perl.

  ($*first***,$***second***,$***third***)=(20,40,60);**

7.8. Mixed Mode Assignments

- Fortran, C, C++, and Perl use coercion rules for mixed-mode assignment.

  int *a***(29); double** *d***=***a***;**