Expressions and Assignment statements

CS 315 – Programming Languages
Pinar Duygulu
Bilkent University
Introduction

• Expressions are the fundamental means of specifying computations in a programming language
• Syntax of Expressions – BNFs
• Semantic of Expressions – will be discussed in this chapter
• To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation
• Essence of imperative languages is the dominant role of assignment statements
Arithmetic expressions

• Arithmetic evaluation was one of the motivations for the development of the first programming languages
• Arithmetic expressions consist of operators, operands, parentheses, and function calls
Arithmetic expressions : Design Issues

– operator precedence rules
– operator associativity rules
– order of operand evaluation
– operand evaluation side effects
– operator overloading
– mode mixing expressions
Arithmetic expressions: Operators

• A **unary** operator has one operand
• A **binary** operator has two operands
• A **ternary** operator has three operands
The *operator precedence rules* for expression evaluation define the order in which “adjacent” operators of different precedence levels are evaluated:

- $3 + 4 * 5$ (35 or 23)

Typical precedence levels:
- parentheses
- unary operators
- ** (if the language supports it)
- *, /
- +, -

Usually Unary minus (-) should not be adjacent to another operator.
- Example, $A+\neg B\neg C$ is usually illegal. It is legal in C.

APL has a single level of precedence rules.
Arithmetic Expressions: Operator Associativity Rule

• The **operator associativity rules** for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated.

• A - B + C - D

• If all + and - are at the **same precedence level** then what is the order of evaluation?

• Typical associativity rule – Left to right

• Exceptions
  – In **FORTRAN**, exponentiation (****) is right associative
  – In Ada, exponentiation (***) is nonassociative; A**B**C is illegal.
  – In C: prefix ++, prefix --, unary +, unary – and = are right associative.
    – Sometimes unary operators associate right to left (e.g., in FORTRAN)

• APL is different; all operators have equal precedence and all operators associate right to left
  • A * B + C is evaluated as A * (B + C)

• Precedence and associativity rules can be overridden with parentheses
Arithmetic Expressions: Parantheses

- Precedence and associativity rules can be altered by placing parantheses
- Example: \((A+B) \times C\)
Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:
    
    ```
    average = (count == 0)? 0 : sum / count
    
    - Evaluates as if written like
      
      ```
      if (count == 0) average = 0
      else average = sum / count
    ```
Arithmetic Expressions: Operand Evaluation Order

• Variables: fetch the value from memory
• Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
• Parenthesized expressions: evaluate all operands and operators first

• If the operands do not have side effects then the operand evaluation order does not matter.
Arithmetic Expressions: Potentials for 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; e.g., for a parameter change:
    ```
a = 10;
/* assume that fun changes its parameter */
b = a + fun(a);
```
Arithmetic Expressions: Side Effects

- Side effects of a function call (**functional side effect**):
- Function changes either one of its parameters or a global variable.
Arithmetic Expressions: Side Effects

- Example in PASCAL:
  - function foo (var x: real): real;
  - begin
  - x := x/2; /* similar situation occurs when */
  - foo := x; /* function changes a global var */
  - end;
  - ...
  - A := 10;
  - B := A + foo(A);
  - ...

- If \( A \) is fetched first, then \( \text{foo}(A) \) is evaluated, the result is 15
- If \( \text{foo}(A) \) is evaluated first, then the result is 10
Arithmetic Expressions: Side Effects

• Example in C:
• int a = 5;
• int foo(){
  • a=7;
  • return 3;
} /* foo */
• main () {
  • a = a + foo();
  • printf("a: %d
", a);
  • }

• When compiled with gcc prints a:10
• When compiled with cc prints a:8
Functional Side Effects

- Two possible solutions to the problem
  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 two-way parameters and non-local references
  2. Write the language definition to demand that operand evaluation order be fixed (E.g., Java: left to right)
     - **Disadvantage**: limits some compiler optimizations
Overloaded Operators

• Use of an operator for more than one purpose is called *operator overloading*
• Some are common (e.g., + for int and float)
• Some are potential trouble (e.g., * in C and C++)
  – Loss of compiler error detection (omission of an operand should be a detectable error)
  – Some loss of readability
  – Can be avoided by introduction of new symbols (e.g., Pascal’s **div** for integer division)
Overloaded Operators

- In C: `&` as a binary operator: bitwise logical AND
- as a unary operator: address of a variable.
- Two unrelated meanings. **Not readable.**
- Example,
  - `x = z & y`
- If the programmer forgets to type `z` it is
  - `x = & y`
- Compiler cannot detect such error.
Overloaded Operators

• In many PLs: / is both REAL and INTEGER division.
• If both arguments are INTEGER, it is INTEGER division with INTEGER result.
• Assume SUM and COUNT are INTEGER, and AVG is REAL.
• AVG = SUM / COUNT
• SUM / COUNT is computed, result is truncated to INTEGER. Then it is assigned to AVG as a REAL value.
• Solution is to use a different symbol for integer division (e.g., div).
Overloaded Operators

- **Ada, C++, and FORTRAN 90** allow **user defined operator overloading**.
- If + and * are overloaded for matrix data type:
  - $A \times B + C \times D$
  - can be written for
  - `MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))`
- **Potential problems:**
  - Users can define nonsense operations
  - Readability may suffer, even when the operators make sense
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`
Type Conversions – Mixed mode expression

• A mixed-mode expression is one that has operands of different types
• Coercion: an implicit type conversion
• Cast: Explicit type conversion requested by the programmer.

• Type conversions (coercion or cast) are either narrowing or widening.
• Narrowing: convert into a subset
  • (e.g., double to float, float to int)
• Widening: convert into a superset
  • (e.g., float to double, int to float)
Type Conversions – Coercions

- In **FORTRAN77** all coercions are widening.
- For example, if in an expression operands are INTEGER and REAL, then the compiler converts INTEGER to REAL type.
- Note that FORTRAN77 does not require the compilers to type check the parameters of user-defined functions.
- Example,

```
INTEGER A, C
FUN (I) = 2 * I  \[ I \text{ is integer} \]
A = 2
D = 3.6
C = FUN (A + D) \[ A+D = 5.6 \text{ is Real} \]
PRINT *, C
END
```
- Prints 10
- In the function call to **FUN**, A is converted to REAL, then \( A + D \) is evaluated as REAL.
- Since **FUN** takes INTEGER argument (I), it is truncated to INTEGER.
- Compiler does not indicate an error.
Type Conversions – coercions

• Disadvantage of coercions:
  – They decrease in the type error detection ability of the compiler
• In most languages, all numeric types are coerced in expressions, using widening conversions
• In Ada, there are virtually no coercions in expressions
• Ada, and Modula-2 do not allow integer and floating-point operands in an expression.
• Exception in Ada: ** (exponentiation operator) can take float or integer as its first argument. The second argument is always integer.
Explicit Type Conversions

- Explicit Type Conversions
- Called *casting* in C-based language
- Both *Modula-2* and *Ada* provide explicit type conversions in the form of *function calls*.

\[
\text{AVG} := \text{FLOAT} (\text{SUM}) / \text{FLOAT} (\text{COUNT})
\]

- Here, \text{SUM} and \text{COUNT} can be any numerical type.
- \text{AVG} is \text{FLOAT}.

  Note that Ada’s syntax is similar to function calls

- In *C*,
- \text{avg} = (\text{float}) \text{sum} / (\text{float}) \text{count};
- In *C++*, both the syntax of Ada and C are acceptable.
Type Conversions: Errors in Expressions

• Causes
  – Inherent limitations of arithmetic
e.g., division by zero
  – Limitations of computer arithmetic
    e.g. overflow
• Often ignored by the run-time system
Relational and Boolean Expressions

- **Relational Operator**: Compares the values of its operands
- **Relational Expression**: Two operands and a relational operator
- The value of a relational expression is **boolean**.
- Operator symbols used vary somewhat among languages (\(!=, /=, .NE., <>\))

<table>
<thead>
<tr>
<th>Operation</th>
<th>Pascal</th>
<th>Ada</th>
<th>C</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>=</td>
<td>=</td>
<td>==</td>
<td>.EQ.</td>
</tr>
<tr>
<td>not equal</td>
<td>&lt;&gt;</td>
<td>/=</td>
<td>!=</td>
<td>.NE.</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>.GT.</td>
</tr>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>.LT.</td>
</tr>
<tr>
<td>greater than or equal</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td>==</td>
<td>.GE.</td>
</tr>
</tbody>
</table>
Relational and Boolean Expressions

• Boolean Expressions
  – Operands are Boolean and the result is Boolean
• Boolean expressions consist of
  – Boolean variables
  – Boolean constants
  – Relational expressions
  – Boolean operators (AND, OR, NOT)
• Example operators

<table>
<thead>
<tr>
<th>FORTRAN 77</th>
<th>FORTRAN 99</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>and</td>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>.OR.</td>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.NOT.</td>
<td>not</td>
<td>!</td>
<td>not</td>
</tr>
</tbody>
</table>

xor
Relational and Boolean Expressions

- C has no Boolean type—it uses int type with 0 for false and nonzero for true.
- One odd characteristic of C’s expressions:
  - `a < b < c` is a legal expression, but the result is not what you might expect:
  - In C, relational operators are left associative.
- `a > b > c` means evaluate first `a>b`, resulting 0 or 1, then compare this result (0, or 1) with `c`.
- That is the result is `1 > c` or `0 > c`, depending on `a>b`.
- Whereas, commonsense interpretation is that `a>b` and `b>c`.
- `main()
  printf ("%d\n", 6>4>1);` prints 0.
- **Readability** requires a PL to include boolean type.
Relational and Boolean Expressions: Operator Precedence

- **Precedence**: NOT (highest) AND OR (lowest)
- Arithmetic, relational and boolean operators can all be in the same expression.
- A PL must define the precedence of all operators.
Relational and Boolean Expressions: Operator Precedence

- Precedence of C-based operators
  
  `prefix ++, --`
  
  `unary +=, -=, prefix ++, --, !`
  
  `*, /, %`
  
  `binary +=, -=`
  
  `<, >, <=, >=`
  
  `=, !=`
  
  `&&`
  
  `||`
Relational and Boolean Expressions: Operator Precedence

In FORTRAN:
Arithmetic (highest)
Relational
Boolean (lowest)

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

is evaluated as

(2 * C)
(A + B)
((A + B) .GT. (2 * C)) (K .NE. 0)
((A + B) .GT. (2 * C)) .AND. (K .NE. 0)
Relational and Boolean Expressions: Operator Precedence

Example:

```fortran
LOGICAL EXPR
   A=2
   B=3
   C=4
   K=5
   EXPR = A + B .GT. 2 * C .AND. K .NE. 0
   PRINT *, EXPR
   A=10
   EXPR = A + B .GT. 2 * C .AND. K .NE. 0
   PRINT *, EXPR
END
```

C Output is
C  F
C  T
C  T

Boolean constants in FORTRAN are `.TRUE. and .FALSE.`
Short Circuit Evaluation

• An expression in which the result is determined without evaluating all of the operands and/or operators

• Example: \((13 \times a) \times (b/13-1)\)
  
  If \(a\) is zero, there is no need to evaluate \((b/13-1)\)

• Problem with non-short-circuit evaluation

  index = 1;
  while (index <= length) && (LIST[index] != value)
    index++;

  When index=length, LIST [index] will cause an indexing problem
  (assuming LIST has length - 1 elements)
Short Circuit Evaluation

- 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 |)
- Most standard Pascal compilers do not use short-circuit evaluation, instead they evaluate all operands.
- Short-circuit evaluation exposes the potential problem of side effects in expressions e.g. \((a > b) || (b++ / 3)\)
- Here, \(b\) is incremented only if \(a<=b\), If the programmer assumes that \(b\) is incremented every time, this is an error.
- In Ada, user can define short-circuit by the and then and or else operators.
- \(I := 1;\)
- \(\textbf{while} (I <= \text{LISTLEN}) \textbf{and then} (\text{LIST}[I] /= \text{key})\)
- \(\text{loop}\)
- \(I := I + 1\)
- \(\text{end loop}\)
- In C and Modula-2 every evaluation of AND and OR is short-circuit.
Assignment Statements

• The general syntax
  \[ \text{<target_var> <assign_operator> <expression>} \]

• The assignment operator
  \[ = \text{ FORTRAN, BASIC, PL/I, C, C++, Java} \]
  \[ := \text{ ALGOLs, Pascal, Ada} \]

• = can be bad when it is overloaded for the relational operator for equality

• In PL/I, the symbol = is both assignemnt and relational operator.

• A = B = C

• assigns A the value TRUE if B = C, FALSE otherwise.
Assignment Statements: Multiple Targets

- In **PL/I**, the statement
- \( A, B = 0 \)
- assigns the value 0 to both \( A \) and \( B \).
Assignment Statements: Conditional Targets

- **Conditional targets (C, C++, and Java)**
  
  \[
  (\text{flag} \ ? \ \text{count1} : \ \text{count2}) = 0
  \]

  Which is equivalent to

  ```
  if (flag)
  count1 = 0
  else
  count2 = 0
  ```

  - Be careful, without the parantheses
    - \text{flag} \ ? \ \text{count1} : \ \text{count2} = 0
    - is equivalent to
      - if \ flag \ then \ count1
      - else \ count2 = 0
Assignment Statements: Compound Operators

• A shorthand method of specifying a commonly needed form of assignment
• Introduced in ALGOL; adopted by C
• Example

\[
a = a + b
\]

is written as

\[
a += b
\]
Assignment Statements: Unary Assignment Operators

- Used in C,
  - `count++;` is equivalent to `count = count + 1;`
- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
  - `sum = ++count;` is equivalent to
    - `count = count + 1;`
    - `sum = count;`
- `sum = count++;` is equivalent to
  - `sum = count;`
  - `count = count + 1;`
Assignment Statements: Unary Assignment Operators

- When two unary operators apply to the same operand, the association is from **right to left**.
- \(-\text{count}++\) is equivalent to \(\text{count} = \text{count}+1; -\text{count};\)
- Unary operations may cause unreadability.
Assignment Statements: Unary Assignment Operators

Consider the following program

```c
main()
{
    int b = 5;
    b = b+++b++; /* legal */
    printf("%d\n", b);
}
```

• The behavior of such a program is not defined clearly in the C language. So, different compilers give different results.
• For example, when compiled with cc it gives 12, whereas
• when compiled with gcc it gives 6.
Assignment as an Expression

• In C, C++, and Java, the assignment statement produces a result and can be used as operands

• An example:

```c
while (((ch = getchar()) != EOF) {...}
```

ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement
Assignment as an Expression

- In C, assignment statement produces a value as if an operator.
- \( a = b = c \)
- \( a \) and \( b \) get the value of \( c \).
- Good for initializing a set of variables to the same value.
- Difficulty:
  - if (\( x=y \))
  - is true if \( y>0 \), and assigns \( y \) to \( x \), but the programmer meant
  - if (\( x==y \))
  - Compiler cannot detect such common typing errors.
Mixed mode assignments

• Assignment statements can also be mixed-mode, for example
  
  ```
  int a, b;
  float c;
  c = a / b;
  ```

• In Pascal, integer variables can be assigned to real variables, but real variables cannot be assigned to integers
• In Java, only widening assignment coercions are done
• In Ada, there is no assignment coercion