A Book Chapter

By Narendra Kumar Chahar

on **Sequence Control**

Index:

Chapter 4

Sequence Control

Sequence control with Expressions:

Ordering is fundamental to most (though not all) models of computing. It determines what should be done first, what second, and so forth, to accomplish some desired task. the language mechanisms used to specify ordering into seven principal categories.

Constructs for specifying the execution order:

1. Sequencing: the execution of statements and evaluation of expressions is usually in the order in which they appear in a program text
2. Selection (or alternation): a run-time condition determines the choice among two or more statements or expressions
3. Iteration: a statement is repeated a number of times or until a run-time condition is met
4. Procedural abstraction: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements
5. Recursion: subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself
6. Concurrency: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor
7. Nondeterminacy: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result

1) Expression Evaluation

An expression consists of

- An atomic object, e.g. number or variable
- An operator applied to a collection of operands (or arguments) that are expressions
- Common syntactic forms for operators:
- Function call notation, e.g. somefunc(A, B, C)
- Infix notation for binary operators, e.g. A + B
- Prefix notation for unary operators, e.g. -A
- Postfix notation for unary operators, e.g. i++

- Cambridge Polish notation, e.g. (* (+ 1 3) 2) in Lisp

"Multi-word" infix, e.g. a>b?a:b in C and myBox displayOn: myScreen at: 100@50 in Smalltalk, where displayOn: and at: are written infix with arguments mybox, myScreen, and 100@50.

Operator Precedence and Associativity

Precedence rules specify that certain operators, in the absence of parentheses, group "more tightly" than other operators. Associativity rules specify that quences of operators of equal precedence group to the right or to the left

• Precedence, associativity

– C has 15 levels - too many to remember

– Pascal has 3 levels - too few for good semantics

– Fortran has 8

– Ada has 6

• Ada puts and & or at same level

– Lesson: when unsure, use parentheses!

Associativity rules are somewhat more uniform across languages, but still display some variety. The basic arithmetic operators almost always associate left to right, so 9 - 3 - 2 is 4 and not 8. In Fortran, as noted above, the exponentiation operator (**) follows standard mathematical convention and associates right-to left ,so 4**3**2 is 262144 and not 4096. In Ada, exponentiation does not associate: one must write either (4**3)**2 or 4**(3**2); the language syntax does not allow the unparenthesized form.

The use of infix, prefix, and postfix notation sometimes lead to ambiguity as to what is an operand of what

- Fortran example: a+b*c**d**e/f
- Operator precedence: higher operator precedence means that a (collection of) operator(s) group more tightly in an expression than operators of lower precedence.
- Operator associativity: determines grouping of operators of the same precedence.
- Left associative: operators are grouped left-to-right (most common)
- Right associative: operators are grouped right-to-left (Fortran power operator **, C assignment operator = and unary minus)
- Non-associative: requires parentheses when composed (Ada power operator **)

Pascal's flat precedence levels are a design mistake.

if A<B and C<D then is grouped as follows

if A<(B and C)<D then

Note: levels of operator precedence and associativity are easily captured in a grammar.

Evaluation Order of Expressions

- Precedence and associativity state the rules for grouping operators in expressions, but do not determine the operand evaluation order!

- Expression a-f(b)-b*c is structured as (a-f(b))-(b*c) but either (a-f(b)) or (b*c) can be evaluated first.
- The evaluation order of arguments in function and subroutine calls may differ, e.g. arguments evaluated from left to right or right to left
- Knowing the operand evaluation order is important
- Side effects: suppose f(b) above modifies the value of b (f(b) has a "side effect") then the value will depend on the operand evaluation order
- Code improvement: compilers rearrange expressions to maximize efficiency, e.g. a compiler can improve memory load efficiency by moving loads up in the instruction stream.

• Side Effects
– often discussed in the context of functions.
– a side effect is some permanent state change caused by execution of function.
• some noticable effect of call other than return value.
• In a more general sense, assignment statements provide the ultimate example of side effects.
– they change the value of a variable
• Several languages outlaw side effects for functions
– easier to prove things about programs
– closer to Mathematical intuition
– easier to optimize
– (often) easier to understand
• But side effects can be nice
– consider rand()
• Side effects are a particular problem if they affect the state used in other parts of the expression in which a function call appears.
– It's nice not to specify an order, because it makes it easier to optimize
– Fortran says it's OK to have side effects
• they aren't allowed to change other parts of the expression containing the function call
• Unfortunately, compilers can't check this completely, and most don't at all

Expression Operand Reordering Issues

- Rearranging expressions may lead to arithmetic overflow or different floating point results
- Assume b, d, and c are very large positive integers, then if b-c+d is rearranged into (b+d)-c arithmetic overflow occurs
- Floating point value of b-c+d may differ from b+d-c
- Most programming languages will not rearrange expressions when parentheses are used, e.g. write (b-c)+d to avoid problems.

Design choices:

- Java: expressions evaluation is always left to right in the order operands are provided in the source text and overflow is always detected
- Pascal: expression evaluation is unspecified and overflows are always detected.
- C anc C++: expression evaluation is unspecified and overflow detection is implementation dependent.
- Lisp: no limit on number representation

Short-Circuit Evaluation

- Short-circuit evaluation of Boolean expressions: the result of an operator can be determined from the evaluation of just one operand
- Pascal does not use short-circuit evaluation
- The program fragment below has the problem that element a[11] is read resulting in a dynamic semantic error:

```
var a:array [1..10] of integer;
...
i := 1;
while i<=10 and a[i]<>0 do
i := i+1
```

- C, C++, and Java use short-circuit conditional and/or operators.
- If a in a&&b evaluates to false, b is not evaluated.
- If a in a||b evaluates to true, b is not evaluated.
- Avoids the Pascal problem, e.g. while (i <= 10 && a[i] != 0) ...
- Ada uses and then and or else, e.g. cond1 and then cond2
- Ada, C, and C++ also have regular bitwise Boolean operators.

References:

[1]. Gray, Susan H. "The effect of sequence control on computer assisted learning." *Journal of Computer-Based Instruction* (1987).

[2]. Mizutani, H., Nakayama, Y., Ito, S., Namioka, Y., & Matsudaira, T. (1992, July). Automatic Programming for Sequence Control. In *IAAI* (pp. 315-331).

[3]. McKeeman, W. M. (1974). Programming language design. In *Compiler Construction* (pp. 514-524). Springer, Berlin, Heidelberg.