Chapter 7: Sequence Control

Principles of Programming Languages

Contents

- Arithmetic Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Selection Statements
- Iterative Statements
- Unconditional Branching

Levels of Control Flow

- Within expressions
- Among program units
- Among program statements

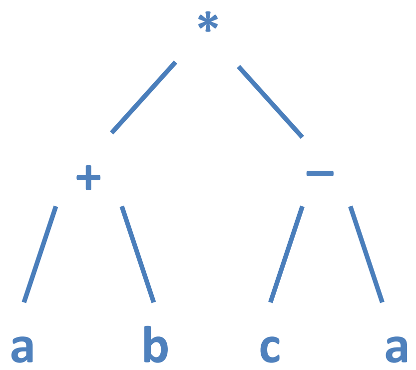
Expressions

- An expression is a syntactic entity whose evaluation either:
  - produces a value
  - fails to terminate → undefined
- Examples

```
4 + 3 * 2
(a + b) * (c - a)
(b != 0) ? (a/b) : 0
```

Expression Syntax

- Expressions have functional composition nature



$(a + b) * (c - a)$

- Common syntax
  - Infix
  - Prefix
  - Postfix

Infix Notation

```
(a + b) * (c – a)
```

- Good for binary operators
- Used in most imperative programming language
- More than two operands?

  `(b != 0) ? (a/b) : 0`
- Smalltalk:

  `myBox displayOn: myScreen at: 100@50`

Precedence

3 + 4 * 5 = 23, not 35

- Evaluation priorities in mathematics
- Programming languages define their own precedence levels based on mathematics
- A bit different precedence rules among languages can be confusing

Fortran	Pascal	C	Ada
		++, -- (post-inc., dec.)	
**	not	++, -- (pre-inc., dec.), +, - (unary), &, * (address, contents of), !, ~ (logical, bit-wise not)	abs (absolute value), not, **
*, /	*, /, div, mod, and	* (binary), /, % (modulo division)	*, /, mod, rem
+, - (unary and binary)	+, - (unary and binary), or	+, - (binary)	+, - (unary)
		<<, >> (left and right bit shift)	+, - (binary), & (concatenation)
.eq., .ne., .lt., .le., .gt., .ge. (comparisons)	<, <=, >, >=, =, <>, IN	<, <=, >, >= (inequality tests)	=, /= , <, <=, >, >=
.not.		==, != (equality tests)	
		& (bit-wise and)	
		^ (bit-wise exclusive or)	
		| (bit-wise inclusive or)	
.and.		&& (logical and)	and, or, xor (logical operators)
.or.		|| (logical or)	
.eqv., .neqv. (logical comparisons)		?: (if...then...else)	
		=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= (assignment)	
		, (sequencing)	

Associativity

- If operators have the same level of precedence, then apply associativity rules
- Mostly left-to-right, except exponentiation operator
- An expression contains only one operator
  - Mathematics: associative
  - Computer: optimization but potential problems

    $10^{20} * 10^{-20} * 10^{-20}$

Parentheses

- Alter the precedence and associativity

  (A + B) * C

- Using parentheses, a language can even omit precedence and associativity rules
  - APL
- Advantage: simple
- Disadvantage: writability and readability

Conditional Expressions

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

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

- C-based languages, Perl, JavaScript, Ruby

Prefix Notation

$$* + a\ b - c\ a$$

$$(* (+ a\ b) (- c\ a))$$

- Derived from mathematical function *f(x,y)*
- Parentheses and precedence is no required, provided the -arity of operator is known
- Mostly see in unary operators
- LISP:

```
(append a b c my_list)
```

Postfix Notation

```
a b + c a – *
```

- Reverse Polish
- Common usage: factorial operator (5!)
- Used in intermediate code by some compilers
- PostScript:

```
(Hello World!) show
```

Operand Evaluation Order

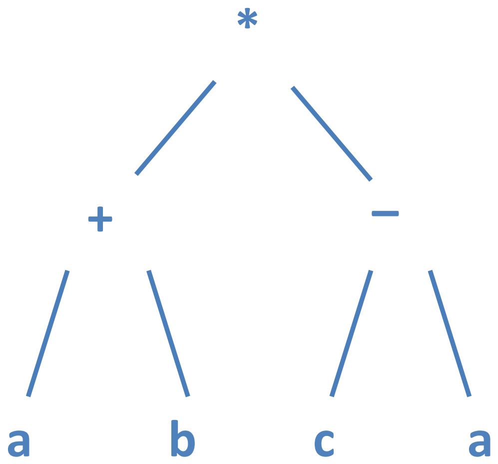
- C program

```
int a = 5;
int fun1() {
   a = 17;
   return 3;
}
void main() {
   a = a + fun1();
}
```

What is the value of a?

- Reason: Side effect!!!

Undefined Operands



- Eager evaluation:
  - First evaluate all operands
  - Then operators
  - How about a `== 0 ? b : b/a`
- Lazy evaluation:
  - Pass the un-evaluated operands to the operator
  - Operator decide which operands are required
  - Much more expensive than eager
- Lazy for conditional, eager for the rest

Short-Circuit Evaluation

```
(a == 0) || (b/a > 2)
```

- If the first operand is evaluated as true, the second will be short-circuited
- Otherwise, “divide by zero”
- How about (a > b) || (b++ / 3) ?
- Some languages provide two sets of boolean operators: short- and non short-circuit
  - Ada: “and”, “or” versus “and then”, “or else”

Statements

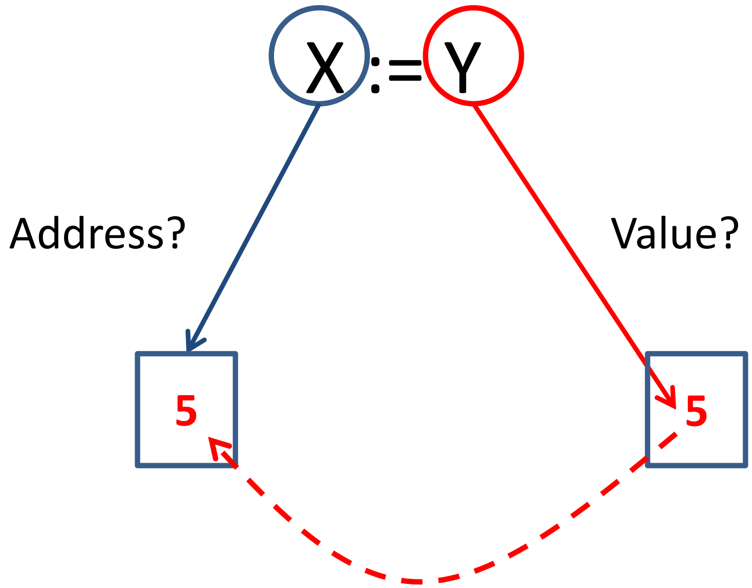
- An expression is a syntactic entity whose evaluation:
  - does not return a value, but
  - have side effect
- Examples:

```
a = 5;
print "pippo"
begin...end
```

Assignment Statements

expr1 OpAss expr2

- Example: Pascal



- Evaluate left or right first is up to implementers

Assignment Statements

- C-based languages consider assignment as an expression

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

- Introduce compound and unary assignment operators (+=, -=, ++, --)
  - Increasing code legibility
  - Avoiding unforeseen side effects

Control Structures

- Control statements
  - Selecting among alternative control flow paths
  - Causing the repeated execution of sequences of statements
- Control structure is a control statement and the collection of its controlled statements

Two-way Selection

```
if control_expression
   then clause
   else clause
```

- Proved to be fundamental and essential parts of all programming languages

Dangling else

```
if (sum == 0)
    if (count == 0)
        result = 0;
else
        result = 1;
```

- Solution: including block in every cases
- Not all languages have this problem
  - Fortran 95, Ada, Ruby: use a special word to end the statement
  - Python: indentation matters

Multiple-Selection

- Allows the selection of one of any number of statements or statement groups
- Perl, Python: don’t have this
- Issues:
  - Type of selector expression?
  - How are selectable segments specified?
  - Execute only one segment or multiple segments?
  - How are case values specified?
  - What if values fall out of selectable segments?

Case Study: C

integer

exact value

```
switch (index) {
   case 1:
   case 3: odd += 1;
           sumodd += index;
           break;
   case 2:
   case 4: even += 1;
           sumeven += index;
           break;
   default: printf("Error in switch").
}
```

- Stmt sequences
- Blocks

Multiple segments exited by break

for unrepresented values

Case Study: Pascal

Integer or character

```
case exp of
    1: clause_A
    2, 7: clause_B
    3..5: clause_C
    10: clause_D
    else clause_E
end
```

- Single statements
- Blocks

multiple values, subrange

Only one segment

for unrepresented values

Iterative Statements

- Cause a statement or collection of statements to be executed zero, one or more times
- Essential for the power of the computer
  - Programs would be huge and inflexible
  - Large amounts of time to write
  - Mammoth amounts of memory to store
- Design questions:
  - How is iteration controlled?
    - Logic, counting
  - Where should the control appear in the loop?
    - Pretest and posttest

Counter-Controlled Loops

- Counter-controlled loops must have:
  - Loop variable
  - Initial and terminal values
  - Stepsize

Case Study: Algol-based

constant

General Form

```
for i:=first to last by step
do
   loop body
end
```

Know number of loops before looping

Semantic

```
   [define i]
   [define first_save]
   [define end_save]
   i = start_save
loop:
   if i > end_save goto out
   [loop body]
   i := i + step
   goto loop
out:
   [undefine i]
```

Case Study: C

General Form

```
for (expr1; expr2; expr3)
  loop body
```

Can be infinite loop

Semantic

```
  expr_1
loop:
  if expr_2 = 0 goto out
  [loop body]
  expr_3
  goto loop
out: . . .
```

Logically Controlled Loops

- Repeat based on Boolean expression rather than a counter
- Are more general than counter-controlled
- Design issues:
  - Should the control be pretest or posttest?
  - Should the logically controlled loop be a special form of a counting loop or a separate statement?

Case Study: C

Forms	**Semantics**

`while` `(ctrl_expr)`
   `loop body`
	`loop:`
	   `if ctrl_expr is false goto out`
	   `[loop body]`
	   `goto loop`
	`out: . . .`

`do`
   `loop body`
`while` `(ctrl_expr)`
	`loop:`
	   `[loop body]`
	   `if ctrl_expr is true goto loop`

User-Located Loop Control

- Programmer can choose a location for loop control rather than top or bottom
- Simple design: infinite loops but include user-located loop exits
- Languages have exit statements: **`break`** and **`continue`**
- A need for restricted goto statement

Case Study: C

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

- What if we replace break by continue?

Iteration Based on Data Structures

- Rather than have a counter or Boolean expression, these loops are controlled by the number of elements in a data structure
- Iterator:
  - Is called at the beginning of each iteration
  - Returns an element each time it is called in some specific order
- Pre-defined or user-defined iterator

Case Study: C#

```
String[] strList = {"Bob", "Carol", "Ted"}:
. . .
foreach (String name in strList)
    Console.WriteLine("Name: {0}", name);
```

Unconditional Branching

- Unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program’s statements
- Dangerous: difficult to read, as the result, highly unreliable and costly to maintain
- Structured programming: say no to goto
- Java, Python, Ruby: no goto
- It still exists in form of loop exit, but they are severely restricted gotos.

Conclusions

- Expressions
- Operator precedence and associativity
- Side effects
- Various forms of assignment
- Variety of statement-level structures
- Choice of control statements beyond selection and logical pretest loops is a trade-off between language size and writability