Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini

Department of Computer Science, Pisa

Lesson 17

- Control Flow
  - Expression evaluation

Control Flow: Ordering the Execution of a Program

- 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

Control Flow: Ordering the Execution of a Program (cont'd)

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. **Exception handling**: when abnormal situations arise in a protected fragment of code, execution branches to a handler that executes in place of the fragment
8. **Nondeterminacy**: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result

Expression Syntax and Effect on Evaluation Order

- 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 ("*mixfix*"), e.g.
    - **a > b ? a : b** in C
    - **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

- 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**

    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 parenthesis when composed (Ada power operator **)

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)	

Precedence levels

Operator precedence levels and associativity in Java

Operatore	Descrizione	Associa a
_ . _	dot notation	sinistra
_ [_]	accesso elemento array	
_ (_)	invocazione di metodo	
_ ++	incremento postfisso	
_ --	decremento postfisso	
++ _	incremento prefisso	
-- _	decremento prefisso	
! _	negazione booleana	
~ _	negazione bit-a-bit	
+ _	segno positivo (nessun effetto)	
- _	inversione di segno	
(*Tipo*) _	cast esplicito	
new _	creazione di oggetto	
_ * _	moltiplicazione	sinistra
_ / _	divisione o divisione tra interi	sinistra
_ % _	resto della divisione intera	sinistra
_ + _	somma o concatenazione	sinistra
_ - _	sottrazione	sinistra
_ << _	shift aritmetico a sinistra	sinistra
_ >> _	shift aritmetico a destra	sinistra
_ >>> _	shift logico a destra	sinistra
_ < _	minore di	sinistra
_ <= _	minore o uguale a	sinistra
_ > _	maggiore di	sinistra
_ >= _	maggiore o uguale a	sinistra
_ == _	uguale a	sinistra
_ != _	diverso da	sinistra
instanceof	appartenenza a un tipo	sinistra
_ & _	AND bit-a-bit	sinistra
_ ^ _	XOR bit-a-bit	sinistra
_ | _	OR bit-a-bit	sinistra
_ && _	congiunzione ‘lazy’	sinistra
_ || _	disgiunzione inclusiva ‘lazy’	sinistra
_ ? _ : _	espressione condizionale	destra
_ = _	assegnamento semplice	destra
_ op= _	assegnamento composto	destra
	(op uno tra *, /, %, +, -, <<, >>, >>>, &, ^, |)	destra

Operator Precedence and Associativity

- C's very fine grained precedence levels are of doubtful usefulness
- Pascal's flat precedence levels is 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 can
  - be captured in a context-free grammar
  - be imposed by instructing the parser on how to resolve shift-reduce conflicts.

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` (that is, `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

Denotational semantics of expressions

- If expressions **don't have side effects**, the semantic interpretation function is

  $\mathcal{E}$: Exp → Env → Store → Eval
- Precedence and associativity rules determine the abstract syntax
- Semantics by structural induction with one rule for each operator, e.g.

  $\mathcal{E}$\{ e1 + e2 \} r s = $\mathcal{E}$\{ e1 \} r s $\oplus$ $\mathcal{E}$\{ e2 \} r s

  where $\oplus$ is the semantic counterpart of +
- If expression **may have side effects**, the function is

  $\mathcal{E}$: Exp → Env → Store → (Eval x Store)
- Order of evaluation of arguments may influence the result
- Semantic rules must specify the order. Eg:
- $\mathcal{E}$\{ e1 + e2 \} r s = let (v', s') = $\mathcal{E}$\{ e1 \} r s in

  let (v'', s'') = $\mathcal{E}$\{ e2 \} r s' in (v' $\oplus$ v'', s'')

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 parenthesis 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 and 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, C++ and Java also have regular bit-wise Boolean operators