Lesson 21

• Control Flow
  – Expression evaluation
  – Structured and unstructured flow
  – Sequencing and selection
Overview

• Expressions evaluation
  – Evaluation order
  – Assignments

• Structured and unstructured flow
  – Goto's
  – Sequencing
  – Selection
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: \( a + b \times c^{d^{e}}/f \) \( a + ((b \times (c^{(d^{e}))})/f) \)
• **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 **)
<table>
<thead>
<tr>
<th>Fortran</th>
<th>Pascal</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>not</td>
<td>++, -- (pre-inc., dec.), +, - (unary), &amp;*, * (address, contents of), !, ~ (logical, bit-wise not)</td>
<td>abs (absolute value), not, **</td>
</tr>
<tr>
<td>*, /</td>
<td>*, /,</td>
<td>* (binary), /, % (modulo division)</td>
<td>*, /, mod, rem</td>
</tr>
<tr>
<td>+, - (unary and binary)</td>
<td>+, - (unary and binary), or</td>
<td>+, - (binary)</td>
<td>+, - (unary)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;&lt;, &gt;&gt; (left and right bit shift)</td>
<td></td>
</tr>
<tr>
<td>.eq., .ne., .lt., .le., .gt., .ge. (comparisons)</td>
<td>&lt;, &lt;=, &gt;, &gt;=, =, &lt;=, IN</td>
<td>&lt;, &lt;=, &gt;, &gt;= (inequality tests)</td>
<td>=, /=, &lt;, &lt;=, &gt;, &gt;=</td>
</tr>
<tr>
<td>.not.</td>
<td></td>
<td>==, != (equality tests)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp; (bit-wise and)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>^ (bit-wise exclusive or)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(bit-wise inclusive or)</td>
<td></td>
</tr>
<tr>
<td>.and.</td>
<td>&amp;&amp; (logical and)</td>
<td>and, or, xor (logical operators)</td>
<td></td>
</tr>
<tr>
<td>.or.</td>
<td></td>
<td></td>
<td>(logical or)</td>
</tr>
<tr>
<td>.eqv., .neqv. (logical comparisons)</td>
<td>?: (if ... then ... else)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>=, +=, -=, *=, /=, %=, &gt;&gt;=, &lt;&lt;=, &amp;=, ^=,</td>
<td>(assignment)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>, (sequencing)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Operator precedence levels and associativity in Java

<table>
<thead>
<tr>
<th>Operatore</th>
<th>Descrizione</th>
<th>Associa a</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ . _</td>
<td>dot notation</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ [ _ ]</td>
<td>accesso elemento array</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ ( _ )</td>
<td>invocazione di metodo</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ ++</td>
<td>incremento postfisso</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ --</td>
<td>decremento postfisso</td>
<td>sinistra</td>
</tr>
<tr>
<td>++ _</td>
<td>incremento prefisso</td>
<td>sinistra</td>
</tr>
<tr>
<td>-- _</td>
<td>decremento prefisso</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ ! _</td>
<td>negazione booleana</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _</td>
<td>negazione bit-a-bit</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ + _</td>
<td>segno positivo (nessun effetto)</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ - _</td>
<td>inversione di segno</td>
<td>sinistra</td>
</tr>
<tr>
<td>( Tipo ) new</td>
<td>cast esplicito</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ * _</td>
<td>moltiplicazione</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ / _</td>
<td>divisione o divisione tra interi</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ % _</td>
<td>resto della divisione intera</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ + _</td>
<td>somma o concatenazione</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ - _</td>
<td>sottrazione</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ &lt;&lt; _</td>
<td>shift aritmetico a sinistra</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ &gt;&gt; _</td>
<td>shift aritmetico a destra</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ &gt;&gt;&gt; _</td>
<td>shift logico a destra</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _ _</td>
<td>minore di</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _ &lt;= _</td>
<td>minore o uguale a</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _ &gt;= _</td>
<td>maggiore di</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _ &gt; = _</td>
<td>maggiore o uguale a</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _ == _</td>
<td>uguale a</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _ != _</td>
<td>diverso da</td>
<td>sinistra</td>
</tr>
<tr>
<td>instanceof</td>
<td>appartenenza a un tipo</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ &amp; _</td>
<td>AND bit-a-bit</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ ^ _</td>
<td>XOR bit-a-bit</td>
<td>sinistra</td>
</tr>
<tr>
<td>_</td>
<td>_</td>
<td>OR bit-a-bit</td>
</tr>
<tr>
<td>_ _ &amp; &amp; _</td>
<td>congiunzione ‘lazy’</td>
<td>sinistra</td>
</tr>
<tr>
<td>_ _</td>
<td></td>
<td>_</td>
</tr>
<tr>
<td>_ _ ? : _</td>
<td>espressione condizionale</td>
<td>destra</td>
</tr>
<tr>
<td>_ _ = _</td>
<td>assegnamento semplice</td>
<td>destra</td>
</tr>
<tr>
<td>_ _ op= _</td>
<td>assegnamento composto</td>
<td>destra</td>
</tr>
</tbody>
</table>
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 are easily captured in a context-free grammar, or can 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
    \[ a - f(b) - b \times c \]
    is structured as
    \[ (a - f(b)) - (b \times c) \]
    but either \( (a - f(b)) \) or \( (b \times c) \) 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
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:
    ```pascal
    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.
    ```pascal
    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
Assignments and Expressions

• Fundamental difference between imperative and functional languages

• **Imperative languages**: “computing by means of side effects”
  – Computation is an ordered series of changes to values of variables in memory (state) and statement ordering is influenced by run-time testing values of variables

• Expressions in (pure) **functional language** are *referentially transparent*:
  – All values used and produced depend on the local referencing environment of the expression
  – A function is *idempotent* in a functional language: it always returns the same value given the same arguments because of the absence of side-effects
L-Values vs. R-Values and
Value Model vs. Reference Model

• Consider the assignment of the form: \( a := b \)
  – The left-hand side \( a \) of the assignment is an \textit{l-value} which is an expression that should denote a location, e.g. array element \( a[2] \) or a variable \texttt{foo} or a dereferenced pointer \( *p \)
  – The right-hand side \( b \) of the assignment is an \textit{r-value} which can be any syntactically valid expression with a type that is compatible to the left-hand side

• Languages that adopt the \textit{value model} of variables copy the value of \( b \) into the location of \( a \) (e.g. Ada, Pascal, C)

• Languages that adopt the \textit{reference model} of variables copy references, resulting in shared data values via multiple references
  – Clu, Lisp/Scheme, ML, Haskell, Smalltalk adopt the reference model. They copy the reference of \( b \) into \( a \) so that \( a \) and \( b \) refer to the same object
  – Most imperative programming languages use the value model
  – \textbf{Java} is a mix: it uses the value model for built-in types and the reference model for class instances
Special Cases of Assignments

• Assignment by *variable initialization*
  – Use of *uninitialized variable* is source of many problems, sometimes compilers are able to detect this but with programmer involvement e.g. *definite assignment* requirement in Java
  – Implicit initialization, e.g. 0 or NaN (not a number) is assigned by default when variable is declared

• Combinations of *assignment operators* (+=, -=, *=, ++, --…)
  – In C/C++ \( a+=b \) is equivalent to \( a=a+b \) (but \( a[i++]+=b \) is different from \( a[i++]=a[i++]+b \), !)
  – Compiler produces better code, because the address of a variable is only calculated once

• *Multiway assignments* in Clu, ML, and Perl
  – \( a,b := c,d \) // assigns \( c \) to \( a \) and \( d \) to \( b \) simultaneously,
    • e.g. \( a,b := b,a \) swaps \( a \) with \( b \)
  – \( a,b := f(c) \) // \( f \) returns a pair of values
Structured and Unstructuctured Flow

- **Unstructured flow**: the use of *goto statements* and *statement labels* to implement control flow
  - Close correspondence with conditional/unconditional branching in assembly/machine code
  - Merit or evil? Hot debate in 1960’s. Dijkstra “GOTO Considered Harmful”
  - Böhm-Jacopini theorem: goto’s are not necessary
  - Generally considered bad: programs are hardly understandable
  - Sometimes useful for jumping out of nested loops and for coding the flow of exceptions (when a language does not support exception handling)
  - Java has no goto statement (supports labeled loops and breaks)
Structured and Unstructured Flow

- **Structured flow:**
  - Statement sequencing
  - Selection with "if-then-else" statements and "switch" statements
  - Iteration with "for" and "while" loop statements
  - Subroutine calls (including recursion)
  - All of which promotes "structured programming"

- Structured alternatives to goto
  - `break` to escape from the middle of a loop
  - `return` to exit a procedure
  - `continue` to skip the rest of the current iteration of a loop
  - `raise (throw)` an exception to pass control to a suitable handler
  - `multilevel return with unwinding` to repair the runtime stack (e.g. `return-from` statement in Common Lisp)

- Cannot jump into *middle* of block or function body
Sequencing

- A list of statements in a program text is executed in top-down order
- A compound statement is a delimited list of statements
  - A compound statement is called a block when it includes variable declarations
  - C, C++, and Java use `{` and `}` to delimit a block
  - Pascal and Modula use `begin ... end`
  - Ada uses `declare ... begin ... end`
- Special cases: in C, C++, and Java expressions can be inserted as statements
- In pure functional languages sequencing is impossible (and not desired!)
- In some (non-pure) functional languages a sequence of expression has as value the last expression’s value
Selection

- If-then-else selection statements in C and C++:
  - `if (<expr>) <stmt> [else <stmt>]`
  - Condition is a bool, integer, or pointer
  - Grouping with `{ and `} is required for statement sequences in the then clause and else clause
  - Syntax ambiguity is resolved with "an else matches the closest if" rule
- Conditional expressions, e.g. `if` and `cond` in Lisp and `a?b:c` in C
- Java syntax is like C/C++, but condition must be Boolean
- Ada syntax supports multiple `elsif's to define nested conditions:
  - `if <cond> then
  <statements>
  elsif <cond> then
  ...
  else
  <statements>
  end if`
Selection (cont’d)

• Case/switch statements are different from if-then-else statements in that an expression can be tested against multiple constants to select statement(s) in one of the arms of the case statement:
  – C, C++, and Java:
    
    ```
    switch (<expr>)
    {
    case <const>: <statements> break;
    case <const>: <statements> break;
    ...
    default: <statements>
    }
    ```
  – A `break` is necessary to transfer control at the end of an arm to the end of the switch statement
  – Most programming languages support a switch-like statement, but do not require the use of a break in each arm
Selection (cont’d)

- The allowed types of `<exp>` depends on the language: e.g. int, char, enum, strings (in C# and Java)
- Some languages admit label ranges
- A switch statement is much more efficient compared to nested if-then-else statements
- Several possible implementation techniques with complementary advantages/disadvantages:
  - Jump tables
  - Sequential testing (like if … then … elseif … )
  - Hash tables
  - Binary search