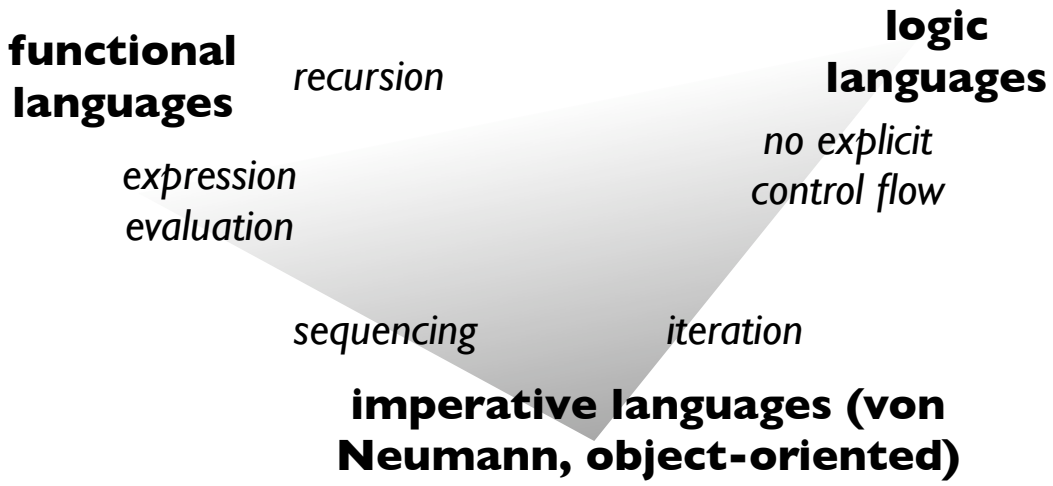
Control Flow

- If the core idea behind naming was, well, *names*, then the core idea behind control flow is *order of execution*
  - Expressions fall under this category somewhat, particularly when it comes to evaluation
- Seven language mechanisms for specifying order: "A programmer who thinks in terms of these categories, rather than the syntax of some particular language, will find it easy to learn new languages, evaluate the tradeoffs among languages, and design and reason about algorithms in a language-independent way."

sequencing	Statement execution order, typically the order in which they appear — but not always!
selection	Run-time choice among two or more statements/ expressions — a.k.a. alternation
iteration	Repeated execution, either a specified number of times or until a run-time condition is true
procedural abstraction	Encapsulation of complex code into single constructs, often subject to parameterization
recursion	Direct or indirect definition of an expression in terms of itself; requires a stack and usually defined via self-referential subroutines
concurrency	Perceived simultaneous execution/evaluation of two or more program fragments
non-determinacy	Deliberately unspecified ordering or choice among statements or expressions — implies that any order will be correct

Control Flow and Language Categories

Certain types of language tend to be associated with certain variations in control flow:



Evolution of Control Flow

Program Type	Control Construct
Machine/assembly language	Conditional or unconditional jumps to addresses
Early imperative languages	*goto* jumps to *statement labels*
"Structured" programming languages	Very limited or total elimination of *goto*

Goto Alternatives

- General approach: instead of statement labels, use lexically-nested constructs (blocks, delimiters)
- First wave: Algol, Pascal
  - Conditional branches — *if…then…else*
  - Enumeration-controlled loops — *for*
  - Logically-controlled loops — *while*

```
      if A .lt. B goto 100
      ...
100:  ...
```

```
if A >= B then
    ...
else
    ...
```

```
      do 100 i = 1, 10, 2
         ...
100:  continue
```

```
for i := 1 to 10 by 2 do
begin
    ...
end;
```

- Second wave: C, Modula
  - Mid-loop exit and continue: *break*, *continue*, *cycle*
  - Mid-subroutine exit: *return*
- Third wave: C++, Java, Ada, ML
  - Error handling (particularly unexpected errors, or errors in deeply nested constructs): *exceptions*
- Issue of *non-local gotos*: prior to exceptions, error-handling *goto* labels are frequently outside a subroutine, sometimes by many levels
- Generalizes to a *continuation*: abstraction of an execution *context*, usable not only with non-local *gotos* but also with subroutine calls and returns, exception handling, call-by-name parameters, iterators, coroutines

Sequencing

- Central to imperative programming — sequencing determines the order in which *side effects* occur
- Straightforward — if a statement precedes another in the source code, then that statement executes before the other at runtime
- Composite design pattern for statements: lists of statements may be delimited into a *compound statement* or *block* that can be used wherever a single statement is expected

Sequencing Design Issues

- Statements vs. expressions or functions — does a statement have a value?
  - For languages that say "yes" to this question, the value of a statement is the value of its last element…in which case statements may be subject to the same considerations discussed previously with expressions
  - Pseudo values: *unit* in ML
- Side effect pros and cons
  - Euclid and Turing explore the prohibition of side effects from functions: facilitates *idempotence* (functions always return the same value when called with the same arguments), simplifying code improvement and reasoning about program behavior
  - But side effects in functions are sometimes very desirable: random number (or other sequence) generators, counters or other global state trackers

Selection

- Generally takes the form of *if…then…else*
  - As seen in the syntax discussion, watch out for nested if/else conditions — generally addressed in the grammar
  - Explicit *elsif* or *elif* keywords avoid nested terminator pile-ups in deep/long conditionals
- Other interesting tweaks include:
  - *unless* variant; switching *if*/*unless* clause and the statement to execute (Perl):

```
go_outside() and play() unless $is_raining;
print "Basset hounds have long ears" if $earLength >= 10;
```

  - Conditionals as part of the language library and not its syntax (Smalltalk):

```
value isNull ifTrue: [ ... ] ifFalse: [ ... ]
“value isNull” evaluates to a Boolean object
```

  - Above, the Boolean class has a method called *ifTrue:ifFalse:*, which takes a code block to execute (expressed as the literal “*[…]*”)

Selection Implementation

- Typical machine translation is a “compare-and-branch” operation, for a limited set of simple comparisons
- Boolean condition would be evaluated cumulatively, with a final comparison at the end
- Short-circuiting allows for *jump code* — compare-and-branch operations *within* the “boolean evaluation” code

```
if ((A > B) and (C > D)) then
    then_clause
else
    else_clause
```

→

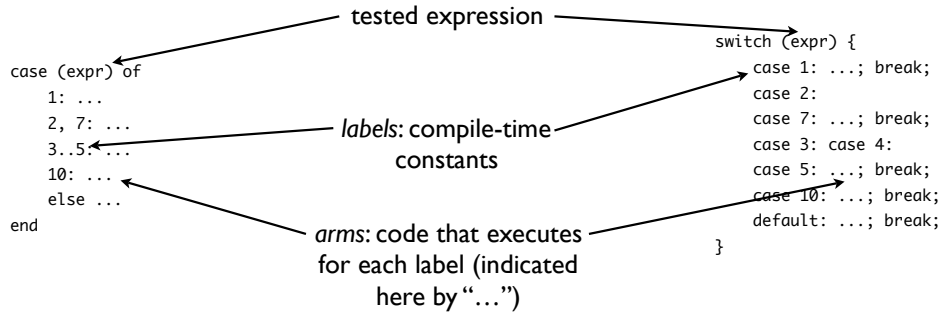
```
r1 := A
r2 := B
r1 := r1 > r2
r2 := C
r3 := D
r2 := r2 > r3
r1 := r1 & r2
if r1 = 0 goto else_clause
```

vs.

```
r1 := A
r2 := B
if r1 <= r2 goto else_clause
r1 := C
r2 := D
if r1 > r2 goto then_clause
```

Case/Switch Selection

Syntactically more concise version of a special nested *if…then…else* case: branches based on a single expression compared to discrete, disjoint ranges of compile-time constants



Interesting variations: Java > 1.5 includes *enum* classes whose instances may be used in a *switch* statement; ML matches *patterns* in function calls

```
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6), VENUS(4.869e+24, 6.0518e6), EARTH(5.976e+24, 6.37814e6),
    MARS(6.421e+23, 3.3972e6), JUPITER(1.9e+27, 7.1492e7), SATURN(5.688e+26, 6.0268e7),
    URANUS(8.686e+25, 2.5559e7), NEPTUNE(1.024e+26, 2.4746e7), PLUTO(1.27e+22, 1.137e6);

    Planet(double mass, double radius) {
        this.mass = mass; this.radius = radius;
    }

    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }

    ... /* surfaceGravity() definition, etc. */
}

...

Planet p = ...;
System.out.println(p.surfaceWeight(otherMass));
switch(p) {
    case MERCURY, VENUS, EARTH, MARS: ...
    case JUPTER, SATURN, URANUS, NEPTUNE: ...
    case PLUTO: ...
}
```

```
(*
 * Helper: r n symbols result => returns the roman
 * equivalent of n appended to result, using only
 * the translations in the mapping called symbols.
 *)
fun r 0 symbols result = result
  | r n [] s = raise Fail "Cannot happen"
  | r n (symbols as (value, rep) :: tail) result =
    if n >= value then
        r (n - value) symbols (result ^ rep)
    else
        r n tail result
```

Switch/Case Design Choices

- Allowable labels: just constants, or allow ranges?
- Single statement or statement list?
  - Note that "single statement" simply means that the switch/case arm must have compound statement delimiters
- Default case: special arm for when the tested expression does not match any label
  - If unsupported, then what happens when no label matches?
  - If supported, then is it required or not? If not required, same question — what happens when no label matches?

Case/Switch Implementation

- Significant difference from "equivalent" nested *if…then…else* implementation
- Instead of evaluations + compare-and-branch, case/switch is implemented as a *jump table*
  - Each label/arm pair is defined as an *offset* from some address, where the offset corresponds to the matching value of the tested expression
  - This is why case/switch generally applies to discrete, scalar types only
- Fast: if *T* is the start of the table and *r* holds the tested expression, a case statement is simply *goto T[r]*
- Space efficiency depends on range of values

Iteration

- Iteration or recursion mean the difference between a finite automaton and a truly useful program
  - Iteration tends to be used more in imperative languages; recursion in functional languages
- Iteration = loops
  - Statements executed repeatedly, primarily for their side effects
- Two primary kinds of loops
  - *Enumeration-controlled*: execute once for every value in a given finite set
  - *Logically-controlled*: execute until some Boolean condition changes value; condition is generally dependent on values that are altered by the loop

Enumeration-Controlled Loops

- First appearance in ForTran:

```
    do 10 i = 1, 10, 2
       ...
10: continue
```

```
    i = 1
L1: ...
    i = i + 2
    if i <= 10 goto L1
```

- Typical components:
  - *index*: the variable that somehow indicates the current iteration in the loop
  - *initial value*: the value of the index for the first iteration
  - *terminal value*: the value of the index after which the loop terminates/exits
  - *step size*: the amount by which to modify the index in each loop iteration
- Syntactic refinement in later languages:

```
FOR i := 1 TO 10 BY 2 DO ... END
```

Enumeration-Controlled Loop Design Issues

- Changes to the loop index, initial and terminal values
  - Prohibited by many languages: evaluated exactly once before the first iteration, so subsequent changes to variables don't affect the loop
- Empty bounds — test for the terminal value before the first iteration
  - Must take step size into account (may be negative) — determines which test to perform before loop starts; explicit *downto* keyword needed if step size is unknown at compile time
- Access to the index outside the loop
  - Value of index after loop terminates? — terminal value or "one past" the terminal value?
  - Scope of index? — some languages restrict the index variable's scope strictly to the loop itself (i.e., loop declaration also serves as declaration of index variable)
- Jumps into and out of a loop — a *goto* issue

Enumeration-Controlled Loop Variations

- Combination loops in Algol 60: single construct for enumeration- and logically-controlled loops

```
for_stmt ::= 'for' id ':=' for_list 'do' stmt
for_list ::= enumerator (',' enumerator)*
enumerator ::= expr | expr 'step' expr 'until' expr | expr 'while' condition
```

- *for* loop in C/C++/Java: not a true enumeration-controlled loop — just syntactic sugar for a special-case logically-controlled loop
- Enumeration control as method instead of distinct language construct (Smalltalk):

```
5 to: 20 by: 2 do: [ :i | ... ]
```

Iterators

Generalization of enumeration-controlled loops: these loops are really iterations over some well-defined set of elements, leading to the alternate *iterator* construct

- Define a set of elements over which to iterate, or a "producer" routine that generates these elements
- Index variable now represents the current element within the loop
- Initial and terminal values, as well as step size, are now determined by the contents of the collection

- "True" iterators
  - Resembles a subroutine that invokes a special *yield* command — *yield expr* sends *expr* to the loop as the value for the next iteration
  - Control bounces between between the loop and the iterator; the loop ends when the iterator "returns" (i.e., no more yields)
  - Present in Clu, Icon, C#, Python (Clu, Icon examples in textbook)
- "Emulated" iterators
  - Sledgehammer: C — completely handcoded
  - Iterator objects: pre-defined data structures that traverse through their members (Euclid, C++, Java, Perl, Smalltalk)

```
// Java < 1.5
for (Iterator it = coll.iterator();
    it.hasNext(); ) {
    Object nextValue = it.next();
    ...nextValue...
}

// Java >= 1.5
for (String s: stringColl) { ...s... }
// ...really just syntactic sugar for
//    pre-1.5 version
```

```
"Smalltalk" "(double quotes delimit
  comments in Smalltalk)"
employees do: [ :emp | ...emp... ].
5 timesRepeat: [ :i | ...i... ].

# Perl
foreach $arg (@ARGV) { ...$arg... }
```

Logically-Controlled Loops

- The overall general case of iteration — in the end, enumeration-controlled loops are special-case versions of logically-controlled loops, and do end up that way in their final machine-level incarnation
- Primary question: when to test the Boolean condition?
  - *Pre-test* (*while*) — test before iteration
  - *Post-test* (*repeat, do while*) — test after iteration; at least one iteration guaranteed
  - *Mid-test* (a.k.a. "one-and-a-half") — test anytime

Mid-Test Logically-Controlled Loop Issues

- Previously an *if…goto* combination in older languages
- *when…exit* (Modula), *break* (C et. al.), *exit…when* (Ada) — requires static semantic check to make sure *exit* is only used inside a loop
- Nested loops: exit to where? — *loop labels* (Java, Perl)

search is an identifier, not a "goto label" !

```
search: for (int i = 0; i < arrayOfInts.length; i++) {
    for (int j = 0; j < arrayOfInts[i].length; j++) {
        if (arrayOfInts[i][j] == searchfor) {
            foundIt = true;
            break search;
        }
    }
}
```

More Fun with Mid-Tests

Perl: *continue* block executes conditionally; these conditions are determined by *next*, *last*, and *redo*

- *next* [label] — proceed to next iteration, with *continue*
- *last* [label] — leave the loop; no *continue*
- *redo* [label] — restart the iteration *without re-evaluating the conditional expression*; no *continue*

```
# This fragment eliminates Perl comments,
# removes trailing backslashes, and bails
# on “stop” by itself.
my $count = 0; my $accum = "";

LINE: while (<STDIN>) {
    next LINE if /^#/;
    last LINE if /^stop$/;
    if (s/\\$//) { redo LINE unless eof(); }
    $accum .= $_;
} continue {
    $count++;
}

# $count will contain the “true” number of
# lines processed, including comments and
# not counting repeats due to backslashes

# $accum will contain the post-processed
# string, without comment lines and terminal
# backslashes
```

Logically-Controlled Loop Odds & Ends

- Logically-controlled loops in Algol 60 used a bogus index variable due to the combo syntax:

```
for i := irrelevant while condition do statement
```

- As mentioned, the *for* construct in C/C++/Java is actually a logically-controlled loop; solves many enumeration-controlled loop design issues, but can’t be optimized as an enumeration-controlled loop
- Smalltalk used its blocks to encapsulate both the conditional expression and the iterated statements:

```
[ loop block ] doWhileTrue: [ condition block ].
```

Recursion

- No extra syntax needed: just allow a function to call itself from its own body (or for multiple functions to call each other cyclically)
- Makes some algorithms easier to write, though not required: recursion and logically controlled iteration have equivalent computational power
- Efficiency depends on implementation
  - Naive implementations tend to favor iteration (say it again: recursion is less efficient with *naive* implementations of iteration and recursion)
  - Certain forms of recursion, such as *tail recursion*, can be very efficient

Tail Recursion

- Primary argument for less efficiency in recursion is the cost incurred by a subroutine call: stack allocation, other bookkeeping — which will always happen if you implement recursion naively (or, "doing what the recursion says, not what it means")
- *Tail recursion* eliminates this overhead: a *tail-recursive function* is a specific form of recursion where no additional computation follows a recursive call; that is, the recursive call, if performed, is the *final computation* in the function

Tail Recursion Example/ Counterexample

```
fun gcd a b =
    (* Assume a, b > 0 *)
    if a = b then
        a
    else
        if a > b then
            gcd (a - b) b
        else
            gcd a (b - a);
```

No calculation after *gcd* call — just return the result of the call

```
fun summation f low high =
    (* Assume low <= high *)
    if low = high then
        f low
    else
        f low + summation f (low + 1) high;
```

Addition still takes place after *summation* recursion returns

- Many compilers (particularly for functional languages) can detect common forms of tail recursion automatically, and transform accordingly
- General case for this transformation passes a *continuation* to the recursive call — recall that a continuation is an encapsulated execution context
- Manual transformations (performed by the programmer) are also possible

```
/* In “compiled” pseudo-code */
gcd(a, b):
start:
    if (a = b) { return a; }
    else if (a < b) {
        b := b - a;
        goto start;
    } else {
        a := a - b;
        goto start;
    }
```

Tail Recursion Helpers

Many non-tail-recursive functions can be transformed using helpers (preferably locally-scoped)

```
fun summation f low high =
    if low = high then
        f low
    else
        f low + summation f (low + 1) high;
```

(note how we rely on addition's associativity — we can accumulate the summation in any order)

```
fun sum f low high = let
    fun sumhelper f low high subtotal =
        if low = high then
            subtotal + f low
        else
            sumhelper f (low + 1) high (subtotal + f low)
in
    sumhelper f low high 0
end;
```

- Arguably, tail recursion isn't "real" recursion — it is actually a rewrite into iterative form
- Some truth to this, with one key distinction — the tail recursive version is *guaranteed* to have *no side effects*
- Consider the iterative and tail-recursive C versions of *gcd*, respectively:

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
    }
}
```

Side effects!

```
int gcd(int a, int b) {
    if (a == b) return a;
    else if (a > b) {
        return gcd(a - b, b);
    } else {
        return gcd(a, b - a);
}
```

No variables are harmed during this tail recursion…

Applicative- vs. Normal-Order Evaluation

- *Applicative-order evaluation*: evaluate all arguments before passing to a subroutine
  - Used by most languages for subroutine evaluations
- *Normal-order evaluation*: evaluate arguments only when needed
  - Used by macros, such as in C:

```
int square(int n) { return n * n; }
```

applicative-order: *n* is bound to a fixed value by the time the multiplication takes place

vs.

```
#define SQUARE(n) ((n) * (n))
```

normal-order: *n* is evaluated each time "n" appears in the macro

- Beware of side effects in normal-order evaluation

```
int x = square(y++);
```

vs.

```
int x = SQUARE(y++); // becomes ((y++) * (y++))
```

- Address this via writing real functions, though you incur subroutine call costs; C++ adds *inline* for function semantics without the subroutine cost
- On the other hand, normal-order may be preferred in unit tests, particularly when testing for failure:

```
// Suppose toRoman() throws an exception.
assertFail(toRoman(-5));
```

- For this, need to delay the potentially failing invocation, so we are "prepared" for the failure (see failure tests for the Roman program in the various languages)

Non-Determinacy

- Already have some kind of non-determinacy with expression evaluation: *f(x) + g(x) + h(x)*
- Guarded command notation [Dijkstra]:

```
if a >= b -> max := a
[] b >= a -> max := b
if
```

  - Any command whose guard is true may execute, but no specification on which will run
  - Variations: at least one guard required to be true? Provide *else* option if no guard is true?
  - Issue: how to choose the guarded command to run — randomization? Circular list/round robin? General guideline is *fairness*: a guard that is true infinitely often should be selected infinitely often

- Useful in conjunction with concurrency (see below)

Guarded Loop Examples

Deterministic	Non-Deterministic

`int gcd(int a, int b) {`
`  while (a != b) {`
`  if (a > b)`
`  a = a - b;`
`  else`
`  b = b - a;`
`  }`
`  return a;`
`}`
	`int gcd(int a, int b) {`
	`  while (a > b) -> a = a - b`
	`  [] (b > a) -> b = b - a;`
	`  return a;`
	`}`

`void server() {`
`  while (true) {`
`  if (read())`
`  processIn();`
`  else if (write())`
`  processOut();`
`  }`
`}`
	`void server() {`
	`  while (read()) -> processIn();`
	`  [] (write()) -> processOut();`
	`  [] true -> /* no-op */ ;`
	`}`