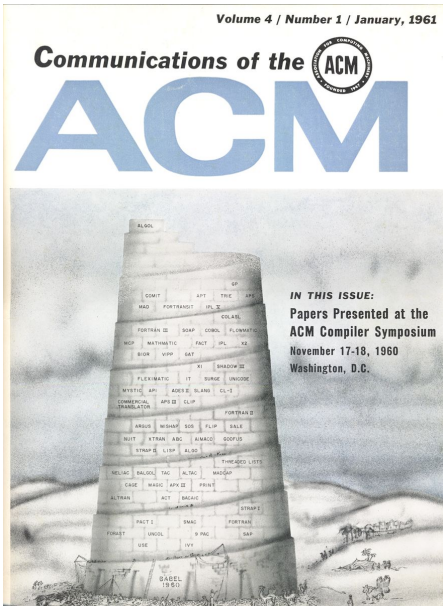
Programming Languages

CSE 3302, Spring 2019
Module 06
Control Flow



M06

Control Flow

Control Flow Introduction

- ① *Sequencing* — Statements are to be executed in a certain specified order. (Usually the textual order in the source file.)
- Sequencing is a central concept in *Imperative* (von Neumann and Object-Oriented) languages.
  - Remember, imperative programs are a sequence of instructions telling the computer ***how*** to carry out a computation.

Control Flow Introduction

- ② *Selection* — Depending on a runtime condition, a *choice* is made of two or more statements or expressions.
- ③ *Iteration* — A section of code is executed repeatedly, usually for a given number of times or until a condition is satisfied.
- ④ *Procedural Abstraction* — A section of code is encapsulated so that it can be handled as a unit, possibly parameterized.

Control Flow Introduction

- ⑤ *Recursion* — A section of code is defined in terms of (a simpler version of) itself, either directly or indirectly.
  - Recursion is a central concept in Functional or Logic languages.
  - Those languages express ***what*** to do, ***not how***, and often in terms of self-referential description.
- ⑥ *Concurrency* — Multiple sections of code are executed "at the same time".
  - Could be on multiple processors or just one processor.

Control Flow Introduction

- ⑦ *Exception Handling* — A section of code is executed *optimistically*, expecting a condition to be true. If not, execution *continues* with a specified section of code.
- ⑧ *Speculation* — A section of code is executed optimistically. If not, another section of code is executed *replacing the first section entirely*.
- ⑨ *Nondeterminacy* — The ordering of a section of code is left purposely unspecified.

Expression Evaluation

- Expressions are simple objects, variables or literal or named constants, or an *operator* or a *function* applied to a collection of *operands*, each of which is an expression.
- *Operators* are special, short names for certain functions.
  - For example, + is the *addition* function in many languages.
  - Can be built in to the language or may be user-defined.
- *Operands* are the arguments to operators.
- Expressions are *evaluated* to produce a *value*.

Operators

- Lots and lots of operators.
- Table goes from highest *precedence* down to lowest.
- Languages try to organize precedence to ease expression writing.

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)	

Operator Precedence

- C's ***15*** levels of operator precedence are tricky to remember.
  - But once one knows and uses them effectively, one hardly ever needs to use parentheses for grouping.
  - Watch out! The shift operators <<, >> are lower precedence than the add + and subtract - operators!
  - So,
    - `1 << 3 + 1 << 5` parses as `(1 << (3+1)) << 5`
    - and ***not*** as `(1 << 3) + (1 << 5)` as one might expect.
  - This is kind of surprising as shifting can be thought of as a multiply-by-2 or divide-by-2.

Operator Precedence

- However, having fewer levels of operator precedence does *not* mean there are no surprises.

- Ada has only ***6*** levels, but the `and` and `or` operators are at the same level of precedence.
  - Hard to believe, but `A or B and C` means `( A or B ) and C` and ***not*** `A or ( B and C )`.

- Pascal's ***4*** levels have `and` at a higher precedence than the comparisons.
  - `A < B and C < D` means `( A < ( B and C ) ) < D` — a static error — and not `( A < B ) and ( C < D )`.

Operator Associativity

- Most operators are *Left Associative*, that is, they bind from left-to-right.
  - `10 - 7 + 3` means `(10 - 7) + 3`, not `10 - (7+3)`.
- Exponentiation is generally *Right Associative*, that is, binding from right-to-left.
  - `2**3**4` means `2**(3**4)`, not `(2**3)**4`.
- Assignment operators are generally *Right Associative*.
  - `a = b = c` means `a = (b = c)`, not `(a = b) = c`.

Expression Evaluation

- Expression evaluation respects the operator associativity and precedence levels.
- Generally the *order* of operand (and function call argument) evaluation is *unspecified*.
- Generally mathematical and logical identities may be (*silently*) applied.
  - `a + b + c` might get evaluated as `c + b + a`.
- However, *parentheses* are *absolutely honored*.
  - So if you want a specific evaluation grouping, parenthesize it!

Expression Evaluation

- Generally all of an operator's operands are evaluated and in no specific order.
- But, *Short-Circuit* operators have a *defined* evaluation order and can leave operands *unevaluated*.
  - A logical `AND` or `OR` operator might be defined to stop evaluating as soon as the answer is known.
  - This permits safer and more compact code.
  - E.g., `pointer != NULL && pointer->field == value` won't evaluate the field value check unless the pointer isn't `NULL`.

Expression Evaluation

- Evaluation order becomes important when expressions have *side effects*.
  - A *side effect* is a change that causes evaluation of the *same* expression to *possibly* have a different result.
  - E.g., in C, the expression `i++` returns the value of `i`, *but also* increments `i` afterwards.
- Functions may be called during expression evaluation and those functions may cause side effects.
- The compiler *doesn't care*! No consideration is given to this.

Side Effects

- Side Effects are *fundamental* to the Imperative / von Neumann model of computing.
  - An assignment is generally executed *solely* for its *side effect*, the change in the value of the variable.
  - A function call may be executed for its *return value*, a *side effect* that it commits, or *both*.
- In (pure) Functional, Logic, Dataflow languages, there are *no* such side effects.
  - Minimizing any required side effects is a goal of such languages.

Side Effects

- Eliminating Side Effects can be helpful.
  - Easier to prove attributes of programs.
  - Closer to mathematical expression.
  - Easier to optimize.
  - Safer.
  - Easier to understand. (Maybe. Depends on the person.)
- Then again, some Side Effects are required.
  - E.g., for I/O, for stateful functions (`random()`), …

Side Effects

- Side Effects are particularly a problem when they affect other parts of an expression being evaluated.
  - E.g., `a[f(i)] = b[f(i)] + c[f(i)]`
  - There's no guarantee of the order of `f(i)` evaluations.
- Since evaluation order is generally unspecified, unpredictable results can occur.
  - Compilers *can't* check this completely, so generally don't even try.

Expression Evaluation

- Operators can be *Infix*, *Prefix*, or *Postfix*.
- Most languages have *Infix* binary operators and *Prefix* unary operators.
  - Binary: `a + b`, Unary: `-c`.
- *Prefix* is known as *Polish Notation*.
- *Postfix* is known as *Reverse Polish Notation*.
- Lisp uses *Cambridge Polish Notation*, a parenthesized version of *Prefix*.
  - E.g., `(* (+ 1 2) 3)`

Jan Łukasiewicz

Logician and philosopher, researcher of propositional logic, the principle of non-contradiction (~(p ∧ ~p)), the law of excluded middle (p ∨ ~p), etc.

https://en.wikipedia.org/wiki/Jan_Łukasiewicz

Assignment

- In imperative languages, *Assignment* is the changing of the value of a location in memory.
- The left-hand side of the assignment indicates the location to change (the *L-Value*), the right-hand side provides the new value (the *R-Value*).
- L-Values are usually variables or parts of variables.
  - A part could be, e.g., a field in a record, an entry in an array.
- R-Values are usually the result of evaluating an expression.

Assignment

- Variables may appear as either L-Values *or* R-Values.
  - E.g., `a = a + 1`
  - On the left, a indicates a *location*.
  - On the right, a indicates a *value*.
- Two models of assignment,
  - *Value model* — Every variable has its *own copy* of the value and assignment means *copying* the value from right to left.
  - *Reference model* — Every variable *refers* to the *same object* in memory and assignment means making the left *refer* to the same object as is on the right.

Values or References?

- Value-oriented languages include C, C++, Pascal, Ada.
  - Remember *pointers* are *values*!
- Reference-oriented languages include most functional languages (Lisp, Scheme, ML, Clu, Smalltalk, etc.)
- Algol is kind of halfway in-between.
- Java is *deliberately* in-between.
  - *Primitive* types are values.
  - *Object* types are references.

Values and References Example in Java

- For primitive type `int`, each variable gets its *own copy*.
  - Assignment to `x` does not affect `y`.
- For object type `ArrayList<>`, variables `a` and `b` *refer* to the *same object*.
  - Altering `a` alters `b` (and vice-versa).

```
int x = 5;
int y = x;

++x;
// y is still 5.
```

```
ArrayList<String> a = new ArrayList<String>();
ArrayList<String> b = a;

a.add( "boo!" );
// now b also has a "boo!" element.
```

"Boxing"

- In languages (such as Java) that have both reference and value types, sometimes it's necessary for the two to work together.
  - E.g., putting an `int` in a `Hashtable`.
- The primitive type value has to be made into an object so it can interoperate with the reference type.
- The process is called *Boxing*.
  - Getting the primitive value back is *Unboxing*.
- Originally, Java required manual boxing/unboxing.
  - More recent versions have made this automatic.

Orthogonality

- A common language design goal is to make the features of a language as *orthogonal* as possible.
  - That is, the language features may be used in any combination with consistent meaning no matter how they are combined.
- Algol 68 made orthogonality a principal design goal.
  - For example, no distinction between *statements* and *expressions*, no distinction between *subroutines* and *functions*.

Expression-Oriented vs. Statement-Oriented

- *Expression-Oriented* languages tend to be Functional, Logic, Dataflow, etc.
  - E.g., in Lisp, there are *no* statements (in the C sense of the word), only expressions.
- *Statement-Oriented* languages tend to be Imperative.
  - C has an assignment *operator* which kind of makes it look like a statement can be in an expression context, but it's really a statement-oriented language.
  - By putting a '`;`' after an expression, it becomes a statement.
    - But, no way to make a statement into an expression (function call?).

Combination Assignment Operators

- Imperative programming languages work via *side effects*, especially of *assignment*.
  - E.g., updating the value of a variable,
    - `a = a + 1`
    - or, horribly, `b.c[3].d = b.c[3].d * 3`
    - Easy to goof up, hard to read, hard to comprehend.
- Solution is to combine operator with assignment.
  - `a += 1` or, even better, `a++`, which makes the intent clear.
  - `b.c[3].d *= 3`, which again makes the intent very clear.
- C provides the prefix and postfix increment / decrement operators as well as combination assignments for all binary operators.

Combination Assignment Operators

- Using the combination assignment operators eliminates any possible confusion about the user's intent.
  - Also cuts down on redundant address calculations at runtime (or factoring costs at compile time).
- Eliminates possible side effect mistakes.
  - First example is dangerous.
  - Second example works, but is redundant.
  - Even better: `A[index(i)] += 1;`
  - Best: `A[index(i)]++;`

```
void update( int A[], int index( int n ) )
{
  int i;

  // calculate i

  A[index(i)] = A[index(i)] + 1;
}
```

```
void update( int A[], int index( int n ) )
{
  int i, j;

  // calculate i

  j = index(i);
  A[j] = A[j] + 1;
}
```

Multiway Assignment

- Also known as *destructuring assignment*, this allows multiple values to be assigned in a single operation.
  - E.g., `a, b = c, d` results in `a = c` and `b = d`.
  - Evaluations of `c` and `d` both take place *before* the assignments.
    - Therefore, `a, b = b, a` *exchanges* `a` and `b` without a temp var.
- Also can be used to return multiple values from function.
  - E.g., `a, b, c = func()`
- Available in Clu, ML, Perl, Python, Ruby, …

Initialization

- Not all imperative languages have an *initialization* mechanism. (For example, Pascal.)
- This is unfortunate as …
  - Using an uninitialized variable is an *extremely* common error.
  - Initializers for *static* variables can save runtime effort.
- *Aggregates* are compile-time initializers for user-defined composite types.
  - E.g., `struct { int i, j; } aVar = { 10, 15 };`
  - Available in C, C++, Ada, Fortran 90, ML, …

Initialization

- Instead of explicit initialization by the user, some languages have initialization by *default values*.
  - C initializes all static or global scope variables as `0`.
  - C# and Java go further, guaranteeing all fields of all objects will be initialized to `0`.
  - Most scripting languages guarantee default values for *all* variables, regardless of scope or lifetime.

Dynamic Checks

- Instead of initialization, a language might create a *dynamic semantic check* for use of an uninitialized variable.
  - Useful because this can reveal *design* errors masked by having default initial values.
- A simple such check can be implemented for Floating Point values by setting them to *Not A Number* (*NaN*).
  - When used in a computation, a hardware exception happens.
- Other types are trickier and more expensive to check.
  - Usually requires extra memory to be allocated and extra checks to be done each time a variable is used.

Definite Assignment

- Java and C# have compile time checks for *Definite Assignment*.
  - The control flow of the program is analyzed and a variable *must* be assigned on each path to where it is used.
- This is an example of a *proof* about the program.
  - Human reasoning might go further, but the compiler has to stick with what it can *prove*.
  - *Undecidable* in the general case.

Java

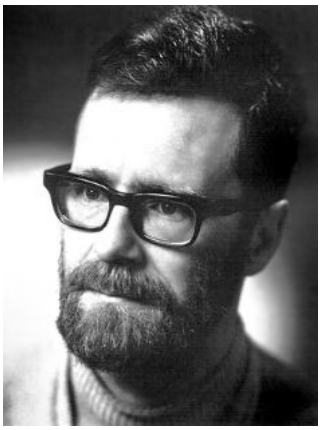
```
int i;
int j = 3;

if ( j > 0 ) {
  i = 2;
}

// Some computation not
// assigning i.

if ( j > 0 ) {
  System.out.println( i );  // ERROR!
}
```

[*"Go To Statement Considered Harmful"*]

- By Edsger W. Dijkstra in *Communications of the ACM* (March 1968).
- Went back and forth for quite a while.
  - ▲ "'GOTO Considered Harmful' Considered Harmful" (March 1987).
  - ▲ "'"GOTO Considered Harmful" Considered Harmful' Considered Harmful?" (May 1987).
  - ▲ Dijkstra entitled his response "*On a Somewhat Disappointing Correspondence*".

[*Edsger W. Dijkstra*]

- Known not only for *Go To Statement Considered Harmful*, but also for *On the Cruelty of Really Teaching Computing Science*.
- … which (i.a.) asserts *Computing Science* is a branch of *Mathematics* and that the *correctness* of programs should be *formally proven*.



Inventor of Dijkstra's Algorithm (shortest paths), Dining Philosophers Problem (shared resources), Guarded Command Language, etc., etc. The ACM annually awards the *Dijkstra Prize* in Distributed Programming. Received the Turing Award (1972)

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

[*"Go To Statement Considered Harmful"*]

- Anti-`goto` won.
  - Ada and C# allow `goto` in only very limited circumstances.
  - Modula, Clu, Eiffel, Java and most scripting languages have no (explicit) `goto` at all.
  - C++ and Fortran 90 have `goto` only for backwards compatibility.
- *Structured Programming* became the hot trend of the 1970s.
  - Much as *Object-Oriented Programming* was for the 1990s.

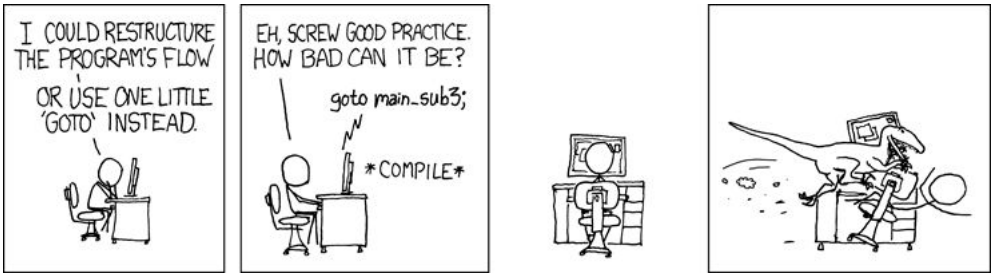
https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

[*Structured Programming*]

- *Structured Programming* emphasized
  - Top-down design (also known as *progressive refinement*).
  - Modularization of code.
  - Structured types.
    - Records, Sets, Pointers, Multidimensional Arrays, …
  - Descriptive variable and constant names.
  - Extensive commenting conventions.
- Almost any well-designed imperative algorithm can be expressed with only *sequencing*, *selection*, and *iteration*.

What's Wrong With `goto`?

- The mantra is, "Just say 'no' to `goto`" as it's thought to lead to hard-to-understand, unmaintainable code.



https://xkcd.com/292/

What's Wrong With `goto`?

- Maybe Dijkstra's comment should have been called "*Go To Statement Considered* ***Unnecessary***".
  - At least after common programming languages were augmented.
- Pretty much any traditional use of `goto` now has a better way to be expressed.
  - `if` … `then` … `else`
  - `for` *or* `while`
  - `switch`
  - `return`
  - `break`, `exit`, `continue`

What's Wrong With `goto`?

- But what about other constructs that are *effectively* `goto`s?
  - E.g., `try` … `except` … `finally`
- Even worse, what about *non-local constructs*?
  - E.g., `try` … `except` … `finally` *with a* `raise` *in a deeply nested call*!
- Considered *OK* because of their generally structured nature.
  - Comment the usage pattern carefully!

```
try :
  # some code that is
  # quite long …

except :
  # some other code
  # that is long …

finally :
  # yet some other code
  # that's way down here
```

Sequencing

- A *Linear* ordering on statements.
  - One statement follows another, usually in the sequence expressed in the textual ordering.
  - If "it doesn't matter" a compiler might reorder statements for, e.g., optimization reasons.
  - If no side effects, either ① or ② gets the same answer.
  - `foo()` and `bar()` might even be executed concurrently.
- The epitome of the *Imperative / von Neumann* style of programming.

```
a = foo();   ①
b = bar();

return a + b;
```

```
b = bar();   ②
a = foo();

return a + b;
```

Selection

- `if` *test* `then` … `else` …
  - Based on *test*, select either `then` or `else` clause.
- `if` … `then` … `else if` … `else if` … `else` …
  - Nesting can cause ambiguities if not properly structured, the *dangling else* problem, as in ①.
    - With which `if` does the `else` clause pair?
  - Generally it's the *closest* `if`.
    - `test2` in this example.
  - Ada uses a specific ending to the `if` to avoid this problem, as in ②.

①
```
if ( test1 ) then
  if ( test2 ) then
    // some code
  else
    // some other code
    // but which then?
```

②
```
if test1 then
   if test2 then
      -- Some statements
   end if
else
   -- Some other statements
end if
```

Compound Statements

- A *compound statement* (or *statement block*) ① is a *grouping* of statements that acts as a *single statement*.
  - Some languages (Pascal, Ada, …) use `begin` / `end`. Others (C, C++, …) use braces { }.
- Execution begins with the first statement in the block, goes through the statements in sequence, and exits following the last statement.
- The problem of nested `if` statements can be addressed with blocks, as in ②.
  - The grouping forces the `else` to pair with the *test1* `if` statement.

①
```
begin
    stmt1
    stmt2
    ...
    stmtn
end
```

②
```
begin
    if test1 then begin
        if test2 then begin
            stmt1
            stmt2
            ...
            stmtn
        end
    end else begin
        stmt1
        stmt2
        ...
        stmtn
    end
end
```

Selection

- Fortran *Assigned* `goto`.
  - A statement label (line number) is assigned to a variable.
  - Later, a `goto` to that variable causes a transfer to the (last) assigned statement label.
  - A common error was to accidently change the variable as an integer between doing the assign and the goto. Undefined results!

```
C     "Assigned" goto
C     200 must be a statement number
      assign 200 to i

C     Now goto wherever i indicates!
      goto i
```

Selection

- Fortran *Computed* `goto`.
  - A *three-way* branching statement based on the value of an integer variable.
  - The *test* is the value of the variable against zero.
  - `< 0`, `= 0`, `> 0` each cause a `goto` to a (potentially) different destination.
    - Statement labels can be repeated.

```
C     "Computed" goto
C       goes to 10 if i < 0
C       goes to 20 if i = 0
C       goes to 30 if i > 0
      goto ( 10, 20, 30 ) i


C       goes to 40 if j <= 0
C       goes to 50 if j > 0
      goto ( 40, 40, 50 ) j
```

Selection

- Fortran *Computed* `goto`.
  - A *three-way* branching statement based on the value of an integer variable.
  - The *test* is the value of the variable against zero.
  - `< 0`, `= 0`, `> 0` each cause a `goto` to a (potentially) different destination.
    - Statement labels can be repeated.

```
C      "Computed" goto
C        goes to 10 if i < 0
C        goes to 20 if i = 0
C        goes to 30 if i > 0
       goto ( 10, 20, 30 ) i


C        goes to 40 if j <= 0
C        goes to 50 if j > 0
       goto ( 40, 40, 50 ) j
```

Selection

- Lisp uses the `COND` special form.
  - Evaluates each test in order until one is found to be non-`NIL`, then returns the value of the corresponding expression.
  - The usual convention is to have the last test be `T` so the final expression is returned.
  - Predecessor to, e.g., C's `switch` statement.

```
(COND
  ((test1) (expr1))
  ((test2) (expr2))
  ...
  ((testn) (exprn))
  (T       (exprf))
)
```

Selection

- `case` / `switch` Statements
  - Often a much cleaner way to implement long `if` / `then` / `else if` / `else if` / `else if` /...
  - Can be implemented by the compiler in many ways
    - Sequential IFs (compiler generated), jump tables, hash tables, binary searching, etc.
    - Depends on the test expression and the allowed values.
  - The C `switch` statement has some odd features.
    - No ranges, no label lists, drop through (requires `break`), …

Iteration

- *Iteration* is the repeated execution of a set of statements.
  - Also known as a *Loop*.
- Iteration comes in two common forms,
  - *Enumeration controlled*, as in ①.
    - Initial value (`1`), Bounding value (`100`), and Step value (`2`).
  - *Logically controlled*, as in ②.
    - Test condition to leave loop.

```
    DO 10 i = 1, 100, 2        ①
    stmt1
    stmt2
    ...
    stmtn
10  CONTINUE            Fortran
```

```
do {                           ②
  stmt1
  stmt2
  ...
  stmtn
} while ( test );            C
```

Enumeration-Controlled Loops

- Essentially all languages have some form of enumeration-controlled loops.
  - Exact syntax and semantics vary considerably by language.
- There are three parts to the control of this kind of loop,
  - The *starting value*, the *ending value* (or *bound*), and the *step*.
  - A count of the number of iterations is easily computable and efficient testing code can be generated.
    - count = max(floor((ending - starting + step) / step), 0)
    - The "max" avoids the case where ending < starting.

Enumeration-Controlled Loops

- Since the number of iterations is generally calculated before the loop starts, changing the starting, ending, and step values *inside* the loop usually has no effects.
- But changing the *loop iteration* (or *index*) *variable* can have *undefined* effects, depending on the language.
- When the loop exits, the index variable generally has the value from the last iteration *plus the step*.
  - Some languages (e.g., Fortran, Pascal) say this value is *undefined*.

Enumeration-Controlled Loops

- With floating point (FP) values, there can be a certain amount of imprecision in the exact value of the loop iteration variable.
  - Fortran originally had only integer enumeration looping.
  - Fortran 77 allowed floating point values.
  - Fortran 90 went back to integer only.
  - The imprecision of FP arithmetic made definitive loop count calculation impossible; values were implementation dependent.

C's `for` loop

- While superficially resembling a enumeration-controlled loop, C's `for` loop is actually a logically-controlled loop.
- It has a *starting* expression, a *test* expression, and a *repeat* expression.
  - *Starting* is executed once, as the loop enters.
  - The *test* is evaluated before each loop iteration. The loop is entered only if the test is true.
  - *Repeat* is evaluated after each iteration completes.
    - Either by falling off the end of the body or via a `continue`.

C's `for` loop

- Since these three clauses are just expressions, there is no specific loop iteration variable and no calculation of an iteration count.
- The user is free to make any changes to any variables at any point in the loop.
- The loop enters each iteration based solely on the truth value of the test expression.

Iterators

- Enumeration-controlled loops can be generalized into the concept of *iterators*.
- Instead of specific numeric starting, bounding, and step values, an iterator enumerates elements of a well-defined (and possibly ordered) set.
- Introduced by Clu, iterators now appear in Python, Ruby, C#, Euclid, C++, Java, Ada, …
  - As usual, there are variations in syntax and exact semantics.

Python Iterator Example

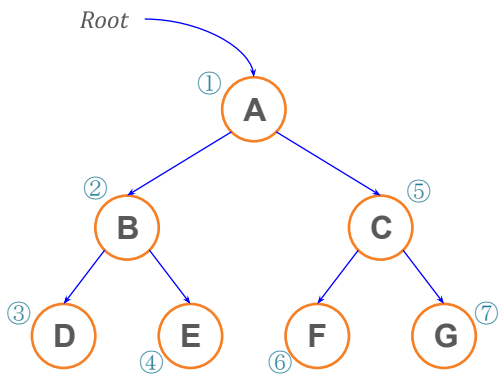
- The `BinaryTree` class has some local info (`data`) as well as a left subtree (`lchild`) and a right subtree (`rchild`).
- The `preorder` method uses `yield` to return any local data, then recursively returns data from the left and right subtrees.
- The use case is quite simple ⟶

```
class BinaryTree :
  def __init__( self ) :
    self.data   = None
    self.lchild = None
    self.rchild = None

  # other methods ...

  def preorder( self ) :
    if self.data is not None :
      yield self.data

    if self.lchild is not None :
      for d in self.lchild.preorder() :
        yield d

    if self.rchild is not None :
      for d in self.rchild.preorder() :
        yield d
```

```
btree = BinaryTree()

# ... build up btree ...

for data in btree.preorder() :
 # do something with data
```

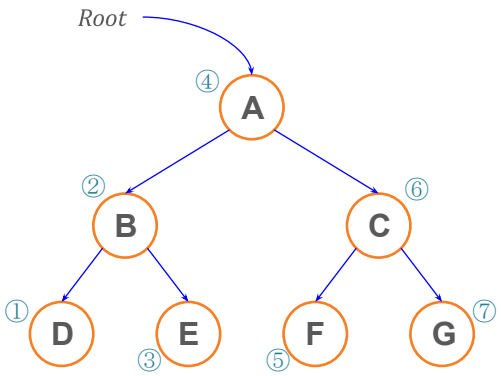
[*Preorder, In Order, Postorder Traversal*]

- Preorder
  - A, B, D, E, C, F, G



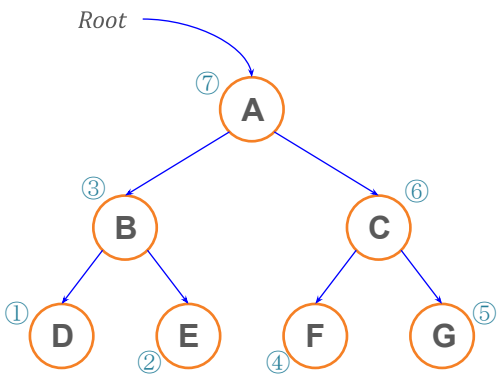
[*Preorder, In Order, Postorder Traversal*]

- In Order
  - D, B, E, A, F, C, G



[*Preorder, In Order, Postorder Traversal*]

- Postorder
  - D, E, B, F, G, C, A



Python Iterator Example

- The `inorder` method is an easy modification of the `preorder` one.
- It recursively returns data from the left subtree, then returns any local data, and finally recursively returns data from the right subtree.

```
class BinaryTree :
  def __init__( self ) :
    self.data   = None
    self.lchild = None
    self.rchild = None

  # other methods ...

  def inorder( self ) :
    if self.lchild is not None :
      for d in self.lchild.preorder() :
        yield d

    if self.data is not None :
      yield self.data

    if self.rchild is not None :
      for d in self.rchild.preorder() :
        yield d
```

Python Iterator Example

- The `postorder` method is also an easy modification of the `preorder` one.
- It recursively returns data from the left and right subtrees, then returns any local data.

```
class BinaryTree :
  def __init__( self ) :
    self.data   = None
    self.lchild = None
    self.rchild = None

  # other methods ...

  def postorder( self ) :
    if self.lchild is not None :
      for d in self.lchild.preorder() :
        yield d

    if self.rchild is not None :
      for d in self.rchild.preorder() :
        yield d

    if self.data is not None :
      yield self.data
```

Logically-Controlled Loops

- Instead of enumerating values, *logically-controlled* loops iterate until / while / unless / depending on a test condition.
- There are many, many variations,
  - `do` … `while`
  - `repeat` … `until`
  - `while` …
  - `loop` …
  - etc.
- The test may occur at the beginning of the loop, at the end, in the middle, or sometimes a combination of places.

Logically-Controlled Loop Styles

```
while ( test ) {
    stmt1
    stmt2
    ...
    stmtn
}
```

```
repeat {
    stmt1
    stmt2
    ...
    stmtn
} until ( test )
```

```
do {
    stmt1
    stmt2
    ...
    stmtn
} while ( test )
```

The simplest such loop is the `while`. The test is checked *before* each iteration of the loop. It's possible that the loop will not be executed even once.

The `repeat` / `until` and `do` / `while` loops check the test at the *end* of each iteration. Each executes the loop body at least once. The `repeat` / `until` loop exits when the condition is *true*; the `do` / `while` exits when the condition is *false*.

```
outer: loop {
   ...
   inner: loop {
       ...
       if ( test1 ) {
         exit inner;
       }
       ...
       if ( test2 ) {
         exit outer;
       }
       ...
   }
   ...
}
```

The generalized `loop` has no specific place for a test. Instead, a regular `if` statement is used and an `exit` statement used to stop the iteration. The `exit` statement can use a label to exit a specific loop instead of the closest enclosing one.

Recursion

- *Recursion* requires no special syntax.
  - Just allow subroutines / functions to call themselves either directly or indirectly (i.e., via other subroutines / functions).
- Iteration and Recursion are *logically equivalent*.
  - *Any* iteration can be rewritten as recursion.
  - *Any* recursion can be rewritten as iteration.
  - Might not be pretty or efficient, but it can be done.

Recursion

- Some languages (e.g., older versions of Fortran) do not permit recursion.
  - The compiler can make static decisions about local storage.
- Some languages (usually the "pure" functional ones) do not have iteration constructs.
  - They aim for the elegance of pure functional design.
- Most languages permit both iteration and recursion.
  - Really comes down to a matter of taste which to use.

Recursion and Iteration Examples

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

// Recursive GCD
int gcd( a, b ) {
  if ( a == b ) {
    return a;
  } else if ( a < b ) {
    return gcd( a, b-a );
  } else {
    return gcd( a-b, b );
  }
}
```

These two examples demonstrate the equivalence of *Recursion* and *Iteration*, at least for the computation of the GCD of two integers and the summation of the value of a function between two limits.

Whether it is *easier* or *more intuitive* to code either operation as recursion or iteration is really a matter of personal preference. The code is not meaningfully different in complexity.

What *can* be different is *efficiency*. A *naive* implementation of recursion is often slower than iteration because of the need for stack frame allocation and deallocation.

```
// Iterative summation
int sum(int f(int i), int low, int high) {
  int total = 0;

  for (int i=low; i<=high; i++) {
    total += f(i);
  }

  return total;
}

// Recursive summation
int sum(int f(int i), int low, int high) {
  if ( low == high ) {
    return f(low);
  } else {
    return f(low) + sum(f, low+1, high);
  }
}
```

Tail Recursion

- Iteration is sometimes thought of as more *efficient* than recursion, but that's *not* necessarily so.
  - *Naive* implementations of recursion can be slow, though.
- A good compiler can make recursion as fast as iteration by recognizing special cases, such as *Tail Recursion*.
  - … and eliminate the need for additional stack frames.
- Tail Recursion is when *no computation* follows the recursive call.
  - The result is whatever the recursive call returns.
- This can be easily turned into an iteration by the compiler.
  - This is trivially so for the GCD case on the previous slide.
  - The summation case is trickier to recognize, but also possible.

Tail Recursion Examples

As originally written, the recursive version of `sum` does an add before returning its value and is thus not tail recursive.

However, it's trivial to convert the sum function to a tail-recursive form simply by introducing an additional (optional) argument `subtotal`.

When the user calls `sum`, this argument is omitted and it takes on the value `0`.

On recursive calls, `subtotal` has the value of the summation so far and so no computation is required after the recursive call returns.

The compiler can now generate an iterative version of the function though it is written as recursive.

```
// Recursive summation
int sum(int f(int i),
        int low, int high) {
  if ( low == high ) {
    return f(low);
  } else {
    return f(low) + sum(f, low+1, high);
  }
}

// Tail-recursive summation
int sum(int f(int i),
        int low, int high,
        int subtotal=0) {
  if ( low == high ) {
    return subtotal+f(low);
  } else {
    return sum(f, low+1, high, subtotal+f(low));
  }
}
```

Nondeterminacy

- *Nondeterminacy* is when the order of execution of a section of code or the selection of an alternative is left deliberately unspecified.
- This occurs in *expression evaluation*,
  - The order of evaluation of arguments to a function call or the operands of an operator is generally unspecified.
- Without special language constructs, however, this generally does *not* occur at the statement level.
  - The user is forced to select an *arbitrary* order of execution.

Nondeterminacy

- Arbitrary orders make correctness proofs more difficult.
  - It's also quite *inelegant*.
- Concurrent programs can often profit from nondeterminacy in that the user is *not* forced to put threads in an order.
  - This can help eliminate the chance of *deadlock*.

Review Questions (1)

1. Name eight major categories of control-flow mechanisms.
2. What distinguishes operators from other sorts of functions?
3. Explain the difference between prefix, infix, and postfix notation. What is Cambridge Polish notation?

Review Questions (2)

6. What is the difference between a value model of variables and a reference model of variables? Why is the distinction important?
7. What is an l-value? An r-value?
8. Why is the distinction between mutable and immutable values important in the implementation of a language with a reference model of variables?
9. Define orthogonality in the context of programming language design.
10. What is the difference between a statement and an expression? What does it mean for a language to be expression-oriented?

Review Questions (3)

11. What are the advantages of updating a variable with an assignment operator, rather than with a regular assignment in which the variable appears on both the left- and right-hand sides?
12. Given the ability to assign a value into a variable, why is it useful to be able to specify an initial value?
13. What are aggregates? Why are they useful?
14. Explain the notion of definite assignment in Java and C#.
15. Why is it generally expensive to catch all uses of uninitialized variables at run time?

Review Questions (4)

16. Why is it impossible to catch all uses of uninitialized variables at compile time?
17. Why do most languages leave unspecified the order in which the arguments of an operator or function are evaluated?
18. What is short-circuit Boolean evaluation? Why is it useful?
19. List the principal uses of goto, and the structured alternatives to each.
20. Explain the distinction between exceptions and multilevel returns.

Review Questions (5)

22. Why is sequencing a comparatively unimportant form of control flow in Lisp?
23. Explain why it may sometimes be useful for a function to have side effects.
25. Why do imperative languages commonly provide a case or switch statement in addition to if... then ... else?

Review Questions (6)

27. Explain the use of break to terminate the arms of a C switch statement, and the behavior that arises if a break is accidentally omitted.
28. Describe three subtleties in the implementation of enumeration-controlled loops.
29. Why do most languages not allow the bounds or increment of an enumeration-controlled loop to be floating-point numbers?

Review Questions (7)

30. Why do many languages require the step size of an enumeration-controlled loop to be a compile-time constant?
33. Does C have enumeration-controlled loops? Explain.

Review Questions (8)

39. What is a tail-recursive function? Why is tail recursion important?