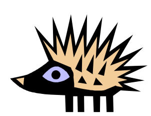
Commands, Control, and Exceptions

Commands

- Building Blocks:
  - **Skips**
  - **Assignments**
  - **Procedure calls**
- Structured Flow:
  - **Sequential/Collateral**
  - **Conditional and iterative commands**
- Sequencers:
  - **Jumps**
  - **Escapes**
  - **Exceptions**
  - **Coroutines**
- Expressions with side-effects



Commands

- Program phrases that can be **executed** in order to update variables or otherwise change the program's *state*
  - Characteristic of *imperative* languages
  - Do not exist in purely functional languages
- **Misnomer**: statements **Why?**
- **Structured *flow*** - single-entry, single-exit
  - **Primitive commands:** skips, assignments, procedure calls
  - **Composite commands:** sequential, collateral, conditional, iterative
    - An imperative language is impoverished if it omits any of the above or is too complicated if it adds to the above, *e.g.* I/O commands
- **Unstructured *flow*** - single-entry, multiple-exit
  - **Sequencers**

Separatists *vs.* Terminists

- **Syntactic issue:** should commands be *terminated* with a special marker (usually semicolon) as the terminists believe, or should they be *separated* by a marker, as the separatists think
- **Separatist approach:** Pascal; annoying problem with if statements…

```
If a > b then
    m := a;
else
    m := b;
```

compilation error

- **Terminist approach:** C
  - When commands are moved around, there is no need to make special arrangements for the last element in the list
- Studies show that programmers tend to make fewer errors in a terminist programming language

Thesis, Antithesis, Synthesis…

- **Hybrid approach:** the last terminator is optional
  - **C initialization sequence:** the only example of compound literals in C

```
int primes[] = {2, 3, 5, 7, 11, 13, };
int months[] = {
            31, 28, 31, 30, 31, 30,
            31, 31, 30, 31, 30, 31 };
```

- **Lax, modern approach:** all separators and terminators are optional
  - **Eiffel:** compiler works harder to understand the program

Skips

- Empty command, dummy command (NOP)
  - written as *";"* in C
  - empty in Pascal
- Useful as components of conditional commands:

  if *E* then *C*

  is an abbreviation for

  if *E* then *C* else skip
- Also as placeholders

Comments

Comments are not command, however it is interesting to examine the syntax of comments in programming languages

- **Line comments:** In Fortran, input lines (punch card images) are designated as comments if they start with a special symbol
- **To EOL comments:** a special comment token designates that the rest of the line is a comment
  - % in TeX
  - -- in Eiffel and Ada
  - # in AWK and CSH
  - // in C++

  Promotes targeted comments
- **Comment blocks:** begin comment and end comment tokens.
  - /* … */ in C
  - (* … *) and { …} in Pascal

Promotes long comments

Main definitional issue: can comments be *nested*?

Assignments

- ***General form:***

  V := E

  In languages where **references to variables** are first class values, V can be an ***expression***. For example in ML, if `m` and `n` are of type `int ref`

  ```
  (if ... then m else n) := 7
  ```

  - In C the above is forbidden, but not in Gnu-C (Gnu-C is a C-like language invented by Richard Stallman)

- ***Kinds of assignments:***

  Multiple assignment:

  ```
  m := n := 0
  ```

  Simultaneous assignment:

  ```
  m, n := n, m
  ```

  Update assignment - combined with a binary operator (C, Algol-68, Icon, Cobol):

  ```
  n +:= 1
  add 1 to N
  ```

Variable Access and Assignments

- A *variable access* can yield:
  - **Content** of the variable
  - **Reference** to the variable
- In Pascal and C, the meaning of an access operation is determined by its context.
- ML: '`val`' *vs.* '`:=`'

```
val n = 5;
> val n = 5 : int
val n = n+1;
> val n = 6 : int
n := n+1;
^
> Type clash...
```

V := E; ML *vs.* Pascal

ML

- both sides are evaluated
- *E* evaluates to *type*
- *V* may be an expression
- *V* must evaluate to *type* `ref`
- '**:=**' is a function with side effect of type **(***type* **ref)** × *type* → ( )
- **dereferencing,** converting ref's to values, must be done explicitly:

```
val n = ref 5;
n := !n + 1;
```

dereferencing

Pascal

- only RHS evaluates
- *E* evaluates to *type*
- *V* must be an identifier
- *V* is of *type*
- '**:=**' is a command
- variable access is context dependent

```
n := n + 1;
```

reference *value*

Procedure Calls

- **A call command:** applying a procedural abstraction to some arguments
- Actual parameters:
  - **Expression**
  - **Variable access:**
    - Variable content
    - Variable reference
- Net effect - update variables of the following kinds:
  - **variables whose reference is passed as actual parameters**
  - **external variables**
  - **static variables**

Sequential and Collateral Commands

- The first of the composite command types
- The order in which commands execute is important
- Sequential command:

  C1; C2;
- Collateral command: no particular order of execution

  `m := 7, n:= n + 1`
- Computation is *deterministic* if it is possible to predict the order in which commands will be executed
- A collateral command is *effectively deterministic* if neither sub-command inspects a variable updated by the other
  - **bad idea:** `n := 7, n:= n + 1`

Conditional Commands

- A number of sub-commands, from which *exactly* one is chosen to be executed
  - **`if`** in most programming languages
  - Pascal and Icon **`case`**
  - Fortran **`if, ifelse`**
  - ... but not C **`switch`**

Why?

- **Collateral conditional command**: the truth valued guard expressions $E_1, E_2, ..., E_n$ are evaluated collaterally.
  If *any* E_i yields true then the corresponding C_i is executed.

```
if
      x >= y then max := x
|
      x <= y then max := y
end if
```
– **effectively deterministic**

So, what's the advantage of this form of writing code?
It is clearer!

  - **Useful for concurrent programming languages**

```
if
      read(inp1, msg) then write(out, msg)
|
      read(inp2, msg) then write(out, msg)
end if
```


Case Commands

- Choose one option, based on a value (usually of an integer, but in Ada any discrete primitive type)
- **case** E **is**

  when v1 => C1

  when v2 => C2

  …

  when vn => Cn

  when others => C0

  end case;
- C's **switch** looks like this, but has a different meaning! All is the same until a value of E is found (say vi) and then Ci is executed, but control then continues with Ci+1.
- To get a regular case, must use **break** at the end of each Ci.

Iterative Commands

- **commands** ***vs.*** **expressions**

procedure calls	**function calls**
conditional commands	**conditional expressions**

- However, iterations (loops) are peculiar to commands

Iterations

- **Indefinite iteration:** number of iterations not determined in advance
  - Testing before loop body: **while**
  - Testing after loop body: **repeat ... until**
- **Definite iteration:** a control sequence based on a *control variable*
  - Usually a sub-range of a discrete ordered primitive type
    - `for i := 1 to 100 do writeln(i);`
    - `for i := 100 downto 1 do writeln(i);`
  - Sometimes an arbitrary arithmetical progression
    - *E.g.,* Fortran's `do 10 i = 100,1,-3`
  - Any set
    - **for** *V* **in** *E* **do** *C*

Collateral Loops

- **Collateral loops:** order of execution of loop body is unspecified
  - **Only makes sense with definite loops**
  - **Useful in parallel and vector computer architectures**
- C `for` loop is indefinite. This is why parallel versions of C are so problematic
- **Collateral loops in AWK:** a definite loop over a set with no particular order
  - The following program prints an unsorted table of all words in the input with the number of times they occur:

```
#!/usr/local/bin/gawk -f
{ #implicit loop over all input lines.
    for (i = 1; i < NF; i++)         #loop over all fields (words) in a line
        count[$i]++;                 #add 1 to cell with index of the i-th field
    }
END {                                #after end of file has reached perform:
    for (word in count)              #collateral loop over all indices of array count
        print word, count[word];
}
```


Control Variable in Definite Iterations

- In many languages, the control variable should be defined outside the loop
- This raises some interesting issues:
  1. **What is its value after termination of the loop?**
  2. **What is its value after a jump out of the loop?**
  3. **What happens if it is changed by the loop body?**
- In Pascal:
  1. **Undefined**
  2. **???**
  3. **Not allowed**
- C++: the loop variable ***can*** be defined in the `for` command, but it will exist after it and can be modified in it
- Algol-68 and Ada: the `for` command is a *declaration* of the control variable which is treated as a constant in each iteration of the loop. The variable ***does not exist*** after termination of the loop!

Generalized Iteration: Power Loops

- **Power loops –** a macro-like iteration structure where the level of nesting is variable

```
nest i:=1 to n
  <commands 1>
  deeper;
  <commands 2>
do
  <commands 3>
end;
```

this is equivalent to this

```
<commands 1> [i←1]
  <commands 1> [i←2]
  …
    <commands 1> [i←n]
      <commands 3>
    <commands 2> [i←n]
  …
  <commands 2> [i←2]
<commands 2> [i←1]
```


Example: The *n* Queens Problem

- Put *n* queens on an *n x n* chessboard with no threats
- A solution using power loops instead of recursion:

```
variable
  Queen: array 1..n of integer;
nest Column := 1  to n
  for Queen[Column] := 1  to n do
    if  OkSoFar(Column) then
          – checks if the Column's queen does not threaten any previous queen
      deeper;
    end;
  end;
do
  write(Queen[1..n]); –  prints valid rows per column
end;
```

Command Expressions

- There are cases in which it is natural to have expressions with loops in them.
  E.g., finding the value of a polynomial
  $a_n x^n + \ldots + a_2 x^2 + a_1 x + a_0$
  (given an array of coefficients) at a point.
- But loops are commands, not expressions!
- A possible solution: *command expressions* – expressions that contain commands
- One example: function body in Pascal and Ada

Command Expressions: Example

A hypothetical Pascal-like command expression for evaluating a polynomial:

```
var  p,r:Real;  i:Integer;
r := DivideBySix(
   begin
    p := a[n];
    for i:=n-1 downto 0 do p:=p*x+a[i];
    yield p;
   end;
   );
```

Expressions with Side-Effects

- In C and many other languages, evaluating an expression can have side effects of updating variables. Expressions take on the form of commands!
- Since functions may have side-effects, even Pascal expressions are commands. In fact, Turbo-Pascal has a flag which allows evaluating expressions as commands.
- Side-effects are considered misleading:

```
if getchar(f) = 'F' then
     gender = female
else if getchar(f) = 'M' then
     gender := male
```

Two *different* characters are read because of the side effects of `getchar`!

- What is the order of evaluating $E_1 \otimes E_2$?
  - **Collateral:** C, Pascal, and Ada (weird imperative, isn't it?)
  - **Left to right:** ML (weird functional, isn't it?) and Icon

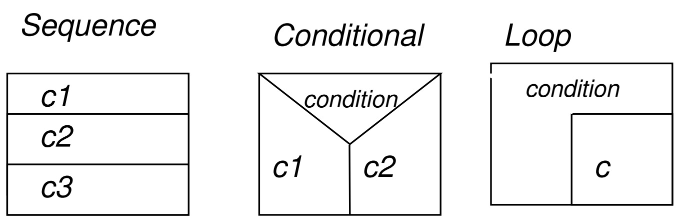
Expression Oriented Languages

- **Expression oriented language:** an *imperative* language in which all distinctions between expressions and commands are eliminated:
  - **Icon**
  - **Algol-68**
  - **C** (to some extent)
  - **BCPL**
  - **ML** (to another extent)
- The value of an assignment is normally that of the updated variable content. But in ML it is the 0-tuple ().
- What is the value of a loop? Usually some neutral value
  - **0, () or &null**
- The main disadvantage – programming style: encourage programming with lots of side-effects
  - **Language designers of Pascal and Ada initially attempted, unsuccessfully, to prohibit all side-effects**

Structured Programming

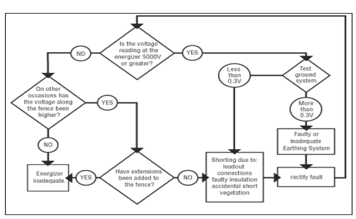
- If the repertoire of control sequences in a program includes only sequences, conditionals (not C switch!) and loops, then the program is called *structured*
  - **Each component has a single entry and a single exit point**
- **Compositional property:** a structured program can be understood and reasoned about by understanding and reasoning about *each component* on its own, and then considering how the components are *composed* together.
- **Nassi-Shneiderman diagrams:** a method for graphically describing structured programs (an alternative to flow charts).

Nassi-Shneiderman diagrams demonstrate the principle of composability



Flow Charts

- An old technique used to describe general control flow:



  - **Every structured program can be described in terms of a flow chart**
  - **However, not every flow chart represents a structured program!**
  - **The fundamental theorem of structured programming (Jacopini-Boehm, 1966):** *there is an algorithm which generates from every flow chart an equivalent structured program*.
    - **The algorithm introduces lots of Boolean auxiliary variables**
- The components of a structured program represent components of a flow chart with a single entry and a single exit point
  - By induction, the flow chart of every structured program has a single entry and a single exit point

Sequencers

- **Sequencer:** a construct that varies the normal flow of control, allowing more general flow charts to be realized by programs
- Single-entry, multiple-exit
- Examples for sequencers:
  - **Jump**: explicit transfer of control from one point to another
  - **Escape**: transfer of control to the end of an enclosing block
  - **Exception**: transfer of control to a *handler* when some condition is met
  - **Coroutine**: end of routine with a possibility to get back to the same place

Jumps

- **Jump**: an explicit transfer of control from one program point to another program point
  - Usually: **`goto`** L where L is a *label*
  - Found in C, Pascal, and many other languages
  - The main control flow mechanism in Fortran (originally)
- **Spaghetti programs:** unstructured programs infected with lots of jumps
  - Backward jumps are often loops in disguise
  - Forward jumps are often conditionals in disguise
- Spaghetti programs are hard to understand because jump commands, unlike most other commands, are not self-explanatory
  - **What is the meaning of** `Goto NextValue` ?

Labels

- Labels denote *program points.* In case of recursion, labels denote a program point in the current activation.
- Literal Labels:
  - **(Meaningful) identifiers:** as in C and most Assembly languages
  - **Numbers:** as in Pascal, Basic and Fortran
    - **In Basic, numerical labels must be in ascending order, and all statements must be labeled.**
- Label declaration:
  - **Required:** as in Pascal (makes their use more difficult and easier to document)
  - **Ad hoc labels:** as in C and Fortran. Makes programs even more difficult to understand.
    - In C, misspelling `default:` would be interpreted as an unused label
  - **Label checking:** is there a jump to every label in the program?
- **Label variables:** the ability to store labels in variables
  - **Obscure feature, unused in modern programming languages**
    - Found in Basic and PL/I, in which one can do perform a GOTO into a label stored in a variable

Restrictions on Jumps

- **Jumps only within a block structure:** Fortran
- **Jumps to any enclosing block:** Algol and Pascal
  - **Labels obey scope rules.** If you can use a variable defined in a nesting block, you can also jump to a label defined in a nesting block (also in C)
  - **No jump to nested blocks**
- **Only jumps within an abstraction (procedure/function def.):**
  - **C allows jumps into a nested block (which cannot be a function)**
  - **C forbids jumping from one function to another**
  - **But so does Fortran, since all blocks in Fortran are abstractions**
- **No jump into a bracketed (compound) command:** Pascal
  - **This includes also the prohibition of making a jump from one subcommand to itself**
- **No jump from a bracketed command into itself:** Pascal
- **No jump into a loop or into a conditional:** C
- **No jumps at all:** Java! (but changes possible in the future: **goto** is a reserved word in Java…)

Problems of Structured Programming

The software paradox: Software is seldom occupied with the problem it is designed to solve. Instead, it works hard to make sure that it is solving the right problem!

A large portion of all software is dedicated to dealing with exceptional, erroneous inputs and funny situations.

- Pascal code tends to be heavily nested and difficult to read:

```
If some error was discovered then Begin
    deal with it
End else Begin
    do a little bit more processing
    If another error was discovered then Begin
        deal with this error
    end else Begin
        continue processing
        If yet another problem has occurred then Begin
            deal with it
        else Begin
            work a little bit more
            If oops, a problem of a different kind was found then Begin
                do something about it
            else Begin
                continue to work
                ...
            end
        end
    end
end
```


Escapes

- **Escape:** Terminate the execution of compound command
  - **A more structured form of jump**
  - **Very useful for making programs clearer**
  - **Makes single entry, multiple exit commands**
- **Examples:**
  - **Escaping out of a loop:** `exit` in Ada and `break` in C
    - **This is the only way of breaking out of a `loop` in Ada**
    - Kinds of loop escapes
      - **Escape the current loop:** break in C
      - **Escape any enclosing loop:** `exit` *L* in Ada and `break` *L* in Perl, where *L* is a loop label
  - **Escaping out of an abstraction:** `return` in C, Fortran, C++, Java
  - **Terminal escape:** terminate the execution of the whole program; `halt` in Fortran, exit() in C
  - **Specialized escape:** `break` out of a `switch` command in C
  - **Universal escape:** escape out of any command. Does not exist in the programming languages we encountered.

Continue

- **Moves on to the next loop iteration**
  - Can be understood in terms of the universal escape

```
while condition do {
    …
    if …then
        continue;
    …
}
```

```
while condition do {
    …
    if … then
        universal_escape;
    …
}
```

  - **Does not increase the number of exit points of the loop**
  - **Impossible to emulate in Pascal, since it is illegal to jump to a compound command from within**
- Kinds of continue:
  - **Inner loop only:** as in C
  - **Any enclosing loop:** as in Perl. Uses a loop label

Emulating Break L in C

```
f() {
    … /* Using break and continue */
    while (--i) {
      while (--k) {
          …
          if (…) goto break2;
      }
      …
      continue;
      break2:
          break;
    }
      …
}
```

```
f() {
        … /* Using return */
        g();
        …
}
g() {
    while (--i) {
        while (--k) {
            …
            if (…) return;
        }
      …
}
```

C Nesting Simplifications

- **Avoid nested conditionals:** eliminate redundant else clauses

```
if (…)
    return | break | continue;
else
    …
```

⇒

```
if (…)
    return | break | continue;
…
```

- **Postpone longer branch:**

```
if (condition) {
    command1;
    …
    command_n;
} else
    single_command;
```

⇒

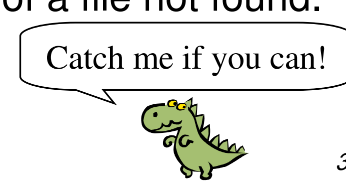
```
if (!condition) {
    single_command;

} else {
    command1;
    …
    command_n;
}
```

- **Introduce auxiliary functions:** as in previous slide

Exceptions

- Abnormal situations or exceptional conditions occur, *e.g.* in
  - **Division by zero**
  - **End of file encountered**
  - **File not found**
  - **Memory exhausted**
- **Robust program:** recovers from exceptions. Done using handlers. 90% of code is sometimes dedicated to these abnormal situations.
- **Detection:** usually at a low level of abstraction. Hardware usually detects division by zero.
- **Recovery:** usually at a high level of abstraction. Only application code can deal with the problem of a file not found.
  - Default “Recovery”: halting the program!



Policies for Exception Handling

- **Resumption**: resume as usual after detection of exception
- **Explicit error handling**: every procedure returns an error code which the invoking procedure has to check
- **Long jump**: moving from a low level of abstraction directly to a higher level of abstraction
- **Lingual constructs**: a handler can be explicitly defined for a command

Resumption

- **Interrupt semantics:** The offending command continues as usual after the handler was invoked
  - **PL/I and Eiffel**
  - **C++: `set_new_handler()`** – gets as argument a pointer to a handler function for failure of a `new` command
- **Pros:** the offending command is neither aware of the problem nor of the solution
- **Cons:**
  - **Confusing semantics:** what is the context of execution of the handler?
  - **Hardly ever works:** in most cases, the handler can *try* to correct the problem, but it cannot be guaranteed that the problem is corrected
    - **Example:** the memory exhaustion handler can free some more memory that is not essential for correct execution (e.g., memory used for caching) or invoke the garbage collector, but it is not guaranteed to be enough
    - Hardly used: **experience shows that resumption policy is not used very often even in languages in which it is implemented**

Explicit Error Handling

- Every procedure returns a specialized error code
  - **In Assembler:** usually the carry flag
  - **In Icon:** each function returns its success value in addition to its real value
  - **C standard usage (library):** 0 or negative value for integers, `Nan` for doubles, null pointer for pointers.
    - **Old C did not allow functions returning structure, so an error value was easy to select**
- The invoking procedure checks this code and tries to recover.
  - Assembler:
    ```
    call proc;
    jc error
    ```
  - **All languages:** heavy responsibility on the programmer
    - **Always remember to test error codes**
    - **Propagate errors up**

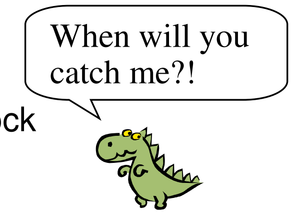
  Most programmers proved to be irresponsible

Long Jump

- **Pascal:** can do a goto to any nesting block
  - No fine control in recursive calls
- **In C:** `setjmp` and `longjmp` allow to jump outside an internally invoked function
    - `setjmp(b)`: saves all CPU registers in buffer `b`
    - `longjmp(b)`: restores the registers from `b`
  - Lots of unsafe properties
- **Rationale:**
  - Error detection is at a low level of abstraction
  - Error handling is at a high level of abstraction

Exception Lingual Constructs

- **C++:** since constructors do not return a value, a constructor's failure must use the "exception" mechanism
- **Ada:** *C* **exception when** *E1* **=>** *H1* **when** *E2* **=>** *H2* …
- **C++ (same semantics but very different syntax):**
  try { *C* **} catch(***E1***) {***H1***} catch (***E2***) {***H2***} ….**
- **Java (similar semantics and syntax):**
  try { *C* **} catch(***E1***) {***H1***} catch (***E2***) {***H2***}**
  …finally … .
- **Exception values:**
  - Unit in Pascal. The result of a `goto` to a nesting block
  - Enumerated type in Ada
  - Any data type in C++



Java Exceptions

- A sequencer **throw** E activates the exception handler associated with the exception E
  - **if (...) throw new** IOException("end of input")
  - Methods declare which exceptions might be thrown
- The exception handling has the form **try** C0
  catch (T1 I1) C1
  …
  catch (Tn In) Cn
  finally Cf

try { rainfall = readannual(); ….} (readannual contains the throw)
catch (IOException e) {System.out.println(e.getmessage() +
" indicates incomplete rainfall data ")}

- The method is not resumed after the handling

Summary on Exceptions

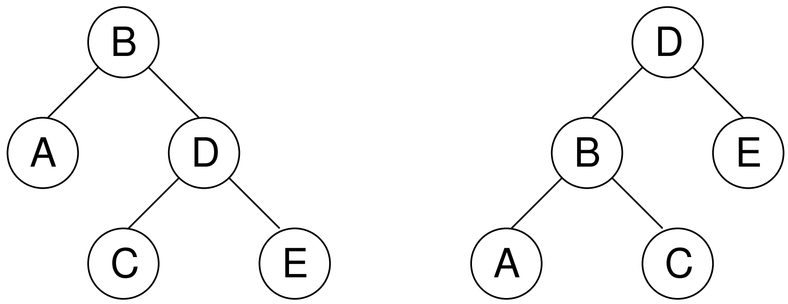
- A serious problem, lots of code is devoted to handling exceptions
- Structured is good, but exceptions seem to require violating basic structured flow of control
- Breaks the regular flow, but can be structured
- Recovery strategy can be fine-tuned to the situation by using handlers
- In Object-Oriented (especially Java), exceptions are defined as objects of a subclass of Exception, and are first-class values

Coroutines

- **Ordinary routine:** performed sequentially from the beginning to the end (which may be a sequencer like `return`) and is never activated again with the same call
- **Coroutine:** a mechanism that allows a routine to pause its execution at a certain point, get back to the calling point, and resume from the same place next time it is called

Motivation for Coroutines

- Compare two binary trees to check if their nodes in in-order traversal are the same:



- Possible solutions without coroutines:
  - Perform two recursive in-order traversals and compare the results (inefficient)
  - Keep a stack for each tree and perform the in-order traversal simultaneously on both trees using *iteration*
- In most programming languages there is no efficient *recursive* solution

A Simula Style Solution

- In Simula, the command **detach** leaves a coroutine defined within an object; **call**ing the same routine in the same object activates the coroutine from the last point **detach** was performed in the routine within that object.

```
class TreeSearch
  MyTree: pointer to Tree;
  CurrentNode: pointer to Tree;
  Done: Boolean;
  procedure Dive (Node: pointer to Tree);
   if Node <> nil then
        Dive(Node^.LeftChild);
        CurrentNode := Node;
        detach;
        Dive(Node^.RightChild);
```

A Simula Style Solution (cont'd)

```
begin -- TreeSearch
   Done := false;
   CurrentNode := nil;
   detach; -- wait for initial values
   Dive(MyTree);
   Done := true;
end;     -- TreeSearch
…
--ASearch,BSearch: pointers to TreeSearch
while not (ASearch^.Done or BSearch^.Done
      or  ASearch^.CurrentNode^.value <>
          BSearch^.CurrentNode^.value)
   do call ASearch^;
      call BSearch^;
   end;
if ASearch^.Done and BSearch^.Done …
      -- trees are equal
```


Coroutines in CLU

- In loops, the control variable can assume successive values from a coroutine called an **iterator**
- An iterator returns values using a **yield** command
- Each time an iterator is called, it is resumed from the location of the last **yield** command it performed
- Example:

```
iterator A() : integer;
   begin
    yield 3;
    yield 4;
   end;
…
for i:=A() do   -- ranges over 3 and 4
   …
End;
```