Statement-Level Control Structures

COS 301: Programming Languages

Topics

- Introduction
- Selection statements
- Iterative statements
- Unconditional branching
- Guarded commands
- Conclusion

Control flow types

- Expression-level:
  - operator precedence
  - associativity
- Statement-level:
  - control statements/structures
- Program unit-level:
  - function calls
  - concurrency

Evolution

- FORTRAN
  - original control statements were simple: conditional branches, unconditional branches, etc.
  - based on IBM 704 hardware
- 1960s: arguments, research about issue
  - Important result: All algorithms represented by flowcharts can be coded using only two-way selection and pretest logical loops
  - I.e., if-then-else and while loops
  - Any language with these features → **Turing-complete**

Goto statement

- Machine level:
  - only have unconditional branches, conditional branches
  - both have form of “goto”
- Gotos: can implement any selection or iteration structure
- But if not careful ⟹ “spaghetti code”
- ⟹ Need help to enforce discipline on control

Control structures

- **Control structure:** control statement + statements it controls
- Control structures ⟹ readability, writability
- Could just have simple control structures
- But maybe not as readable/usable as we’d like

Simple control structures

- E.g., FORTRAN’s IF statement

  IF (*logical-exp*) *stmt*

  - Since there were no blocks in FORTRAN, often led to things like:

—FORTRAN—

```
      IF (A .GT. B) GOTO 10
      stmt1
      stmt2
      GOTO 20
   10 else-stmt
   20 stmt-after-if
```

—pseudocode—

```
if (a <= b) {
stmt1
stmt2
}
else else-stmt
stmt-after-if
```


Simple control structures

- E.g., FORTRAN’s arithmetic IF:

```
      IF (SUM/N – 50) 100,200,300
  100 WRITE (6,*) 'Below average.'
      GOTO 400
  200 WRITE (6,*) 'Average.'
      GOTO 400
  300 WRITE (6,*) 'Above average.'
  400 WRITE (6,*) 'Done.'
```

Simple control structures

- Similarly, iteration constructs were simple:

```
      DO 200 I=1,10,0.5
      WRITE (6,*) 'I=', I, '.'
      IF (I .GT. 9) GOTO 300
      WRITE (6,*) 'Did not exit'
 200  CONTINUE
 300 WRITE (6,*) 'Out of loop.'
```


Structured programming

- Instead of designing control structures based on machine ⟹ design to reflect how humans think
  - more readable
  - more writable
  - reduce spaghetti code

Structured programming

- Structured programming
  - High-level control structures
  - Linear control flow, if consider control structures as statements
  - Usually **top-down design**
- Most languages: high-level control structures

Control structure design

- **Multiple exits** from control structure?
  - Almost all languages allow multiple exits — e.g., Perl:

```
$count = 1;
while ( 1 ) {
 last if ($count > 20);  ←
 $count++;
}
```

  - Question: is target of exit unrestricted?
  - If so, then ⇔ gotos
- **Multiple entry points:**
  - Would need gotos, labels
  - Unwise in any case

Topics

- Introduction
- Selection statements
- Iterative statements
- Unconditional branching
- Guarded commands
- Conclusion

Selection statement

- **Selection statement:** chooses between 2 or more paths of execution
- Categories:
  - Two-way selectors
  - Multi-way selectors

Two-way selection

- E.g., `if` statement

```
<ifStatement> → if (<exp>) <stmt>
                    [else <stmt>]
```

- Design issues:
  - Type of control expression (<exp>)?
  - How are then, else clauses specified?
  - How should nested selectors be specified?

Control expression

- **Syntactic markers:**
  - sometimes `then` or other marker (Python's ":")
  - if not, then enclose <exp> in () — e.g., C-like lang's
- C — no Booleans (more or less), so control expression → integers, arithmetic expressions, relational expressions
- Many languages coerce control expression to Boolean
  - 0 = false, non-zero = true
  - empty string = false, non-empty = true
  - some coerce to integer first, then test
- Other languages: *must* be Boolean (Ada, Java, C#, Ruby…)

Then/else clauses

- Most modern languages: single statements or compound statements
- Most C-like languages: compound statements using {}
- Perl: *all* clauses delimited by {}:

```
if ($x>$y) {
   print "greater\n";
} else {
   print "less\n";
}
```


Then/else clauses

- Fortran 95, Ruby, Ada: statement sequences, delimited by keywords

```
if (<expr>) then
   …
else
   …
end if
```

- Python: indentation

```
if x > y :
   x = y
   print "greater"
…
```


Nesting selectors

- Java:

```
if (sum == 0)
   if (count == 0)
      result = 0;
else result = 1;
```

- The `else` goes with…?
- Java's static semantics rule: `else` matches nearest `if`
- Can force alternative with {}
- Also for C, C++, C#
- Perl: not a problem — all clauses use {}

Selectors using reserved words

- Avoid nested selection issue: use reserved words to end clauses
- E.g., Fortran 95 (previous example)
- E.g.: Ruby:

```
if sum == 0 then
   if count == 0 then
      result = 0
   else
      result = 1
   end
end
```

Nesting selectors

- Python — indentation decides

```
if sum == 0:                  if sum == 0:
    if count == 0:                if count == 0:
      result = 0     vs.            result = 0
    else:                     else:
      result = 1                    result = 1
```


Multi-way selection statements

- Select any number of control paths
- Can use 2-way selector to express multi-way semantics
- Can use multi-way selector to express 2-way semantics
- But better to have both — less clumsy (better readability/writability)

Multi-way selection

- Two different purposes:
  - Single scalar's value ⟹ multiple control paths (ordinal values) → **case/switch statements**
  - Flattening deeply nested if statements consisting of mutually-exclusive cases → else-if statements
- Some languages combine both purposes into a single flexible case statement

Case/switch design issues

- Form & type of control expression?
- How are the **selectable segments** specified?
- Single selectable segment per execution, or multiple?
- Specification of case values?
- What about values not handled by a case?

Case/switch statement

- Selection based on small set of ordinal values
  - Start: FORTRAN's computed GOTO:

```
GO TO (100, 87, 345, 190, 52) COUNT
```

  - Semantics: if count = 1 goto 100, if count = 2 goto 87 etc.
- Can be implemented as a *jump table*

Jump Tables

- "Table" of jump statements in machine code
- Convert value of control expression into index into table
- Goto base of table + index

Case/switch statement

- Case/switch entry statement contains a *control expression*
- Body of statement:
  - multiple tests for values of control expression
  - each with associated block of code
- Control expression needs small number of discrete values → efficient (jump table) implementation

C switch statement

- Control expression: integers only
- Selectable segments: statement sequences or compound statements
- Any number of segments can be executed — no implicit branch at end of segment (have to use **break**)
- Default clause: unrepresented values
- If no default and no selectable segment matches → statement does nothing
- Statement designed for flexibility
  - However, flexibly much greater than usually needed
  - Need for explicit break — seems like a design error
  - May lead to poor readability

Example for C-like

```
switch(n) {
  case 0:
    printf("You typed zero.\n");
    break;
  case 1:
  case 9:
    printf("n is a perfect square\n");
    break;
  case 2:
    printf("n is an even number\n");
  case 3:
  case 5:
  case 7:
    printf("n is a prime number\n");
    break;
  case 4:
    printf("n is a perfect square\n");
  case 6:
  case 8:
    printf("n is an even number\n");
    break;
  default:
    printf("Only single-digit numbers are allowed\n");
  break;
}
```


C# changes to switch

- C# — static semantics rule disallows the implicit execution of more than one segment
- Each segment must end with unconditional branch — **goto**, **return**, **continue**, **break**
- Control expression, case constants can be strings

C# syntax

```
switch (expression)
{
   case constant-expression:
      statement
      jump-statement
   [default:
      statement
      jump-statement]
}
```


C# example

```
switch (value){
case -1:
     minusone++;
     break;
case 0:
     zeros++;
     goto case 1;
case 1:
     nonnegs++;
     break;
default:
     return;
}
```

Ada case statement:

- Expression: any ordinal type
- Segments: single or compound
- Only one segment executed out of choices
- Unrepresented values not allowed (have default keyword, though)
- Constant list forms:
  - constant
  - list of constants
  - subranges
  - Boolean OR operators

Ada case statement syntax

```
case expression is
    when choice_list => stmt_sequence;
    …
    when choice_list => stmt_sequence;
    when others => stmt_sequence;
end case;
```

- More reliable than C's switch — once segment selected and executed → statement after case

Ada case example

```
type Directions is (North, South, East, West);
Heading : Directions;
case Heading is
  when North =>
     Y := Y + 1;
  when South =>
     Y := Y - 1;
  when East =>
     X := X + 1;
  when West =>
     X := X - 1;
end case;
```

Ada also supports choice lists:

```
case ch is
     when 'A'..'Z'|'a'..'z' =>
```


Ruby's switch statement

```
case n
when 0
   puts 'You typed zero'
when 1, 9
   puts 'n is a perfect square'
when 2
  puts 'n is a prime number'
  puts 'n is an even number'
when 3, 5, 7
   puts 'n is a prime number'
when 4, 6, 8
   puts 'n is an even number'
else
   puts 'Only single-digit numbers are allowed'
end
```

Ruby's switch statement

- Switch can also return a value in Ruby:

```
catfood = case
             when cat.age <= 1 then junior
             when cat.age > 10 then senior
             else                  normal
           end
```


Perl, Python, Lua

- Perl, Python and Lua do not have multiple-selection constructs — but can do same thing with else-if structures
- Python: use `if…elif…elif…else`

Perl, Python, Lua

- Perl has a *module*, Switch, that adds a switch statement when used:

```
use Switch;

$var = 10;
@array = (10, 20, 30);
%hash = ('key1' => 10, 'key2' => 20);

switch($var){
   case 10           { print "number 100\n"; next; }
   case "a"          { print "string a" }
   case [1..10,42]   { print "number in list" }
   case (\@array)    { print "number in list" }
   case (\%hash)     { print "entry in hash" }
   else              { print "previous case not true" }
}
```

When the above code is executed, it produces following result:
number 100
number in list

From http://www.tutorialspoint.com/perl/perl_switch_statement.htm

Lisp

- Has both kinds of multi-way conditionals
- **case** statement:

  (case foo
    (*valSpec stmt…*)
    (*valspec stmt…*)
    …
    (otherwise *stmt…*)
- "otherwise" clause optional
- Ex:

  (case (read)
      ((#\y #\Y) 'ok)
      ((#\n #\N) 'nope)
      (otherwise (error "Bad response!")))

Lisp

- **cond** statement
- Syntax:

  (cond (*test* {stmt}*)*)
- Semantics:
  - first clause whose test is non-nil executes
  - return last form evaluated
  - if no clause's test is true: return nil
- Ex:

```
(defun factorial (n)
  (cond
   ((not (numberp n)) (warn "bad argument ~s" n)
                      nil)
   ((<= n 1) 1)
   (t (* n (factorial (1- n))))))
```


Implementing Multiple Selection

- Four main techniques

1. Multiple conditional branches

```
mov eax, var
cmp eax, 1
je target1
cmp eax, 2
je target2
…
```

2. *Jump tables*
3. Hash table of segment labels
4. Binary search table

Implementing Multiple

- Four main techniques

1. Multiple conditional branches
2. Jump table

   (a) Constructed in program code (above)

   (b) Indexing into array

```
mov edx, var
mov edi, jmptable_address
jmp [edi+edx]
```

3. Hash table of segment labels
4. Binary search table

Implementing Multiple

- Four main techniques

1. Multiple conditional branches
2. Jump tables
3. Hash table of segment labels
4. Binary search of table

Implementing Multiple

- Four main techniques

1. Multiple conditional branches
2. Jump tables
3. Hash table of segment labels
4. Binary search of table

Deeply-nested ifs

```
if (grade > 89) {
      ltr = 'A';
} else {
      if (grade > 79) {
            ltr = 'B';
      } else {
            if (grade > 69) {
                  ltr = 'C';
            } else {
                  if (grade > 59) {
                        ltr = 'D';
                  } else {
                        ltr = 'E';
                  }
            }
      }
}
```


Using else-if statement

```
if (grade > 89) {
    ltr = 'A';
} else if (grade > 79) {
    ltr = 'B';
} else if (grade > 69) {
    ltr = 'C';
} else if (grade > 59) {
    ltr = 'D';
} else
    ltr = 'E';
}
```


Multi-way selection with `if`

- Else-if and similar statements/clauses ⟹ multi-way selection
- E.g. Python's elif:

```
if count < 10:
  bag1 = True
elif count < 100:
  bag2 = True
elif count < 1000:
  bag3 = True
```

Multi-way selection with `if`

- Can be rewritten as (e.g.) a Ruby case statement:

```
case
  when count < 10 then bag1 = true
  when count < 100 then bag2 = true
  when count < 1000 then bag3 = true
end
```


Topics

- Introduction
- Selection statements
- Iterative statements
- Unconditional branching
- Guarded commands
- Conclusion

Iterative statements

- Repetition in programming languages:
  - **recursion**
  - **iteration**
- First iterative constructs — directly related to array processing
- General design issues:
  - How is iteration controlled?
  - Where is the control mechanism in the loop?

Loop Control

- **Body**: collection of statements controlled by the loop
- Several varieties of **loop control**:
  - Test at beginning (while)
  - Test at end (repeat)
  - Infinite (usually terminated by explicit jump)
  - Count-controlled (restricted while)

Count-controlled loops

- **Counting iterative statement:**
  - loop variable, means of specifying initial, terminal, and step values (loop parameters)
  - e.g., **for** statement
- Note — some machine architectures directly implement count controlled loops (e.g., Intel LOOP instruction)
- Design issues:
  - What are the type and scope of the loop variable?
  - Can the loop variable be changed in the body? If so, how does it affect loop control?
  - Loop parameters — evaluate only once, or each time through the loop

Fortran 95 DO Loops

- FORTRAN 95 syntax

```
DO var = start, finish[, stepsize]
   …
END DO
```

- Stepsize: any value but zero
- Parameters can be expressions
- Design choices:
  - Loop variable must be INTEGER
  - Loop variable cannot be changed in the loop
  - Loop parameters are evaluated only once
  - Parameters can be changed within loop — but evaluated only once, so no effect on loop control

Operational semantics

```
init_val = init_expression
term_val = terminal_expression
step_val = step_expression
do_var = init_val
it_count = max(int(term_val – init_val + step_val)/step_val,0)
loop:
    if it_count <= 0 goto done
    [body]
    do_var = do_var + step_val
    it_count = it_count - 1
    goto loop:
done:
```


Example: Ada `for` loop

- Ada

```
for var in [reverse] discrete_range loop
  ...
end loop
```

- Design choices:
  - Loop variable → discrete range
  - Loop variable does not exist outside the loop
  - Cannot change loop variable in loop
  - The discrete range is evaluated just once
  - Cannot branch into the loop body

C-style Languages

- C-based languages

```
for ([expr_1] ; [expr_2] ; [expr_3])
    statement
```

  - All expressions are optional
  - Expressions:
    - Can be multiple statements, separated by commas
    - Value of list of expressions is value of last expression

C-Style For Loops

- This…

```
for (expressions1;  expression2; expressions3)
  statement;
```

- …is semantically equivalent to:

```
expressions1;
while (expression2) {
  statement;
  expressions3;
}
```


C-style **for** loops

- Consider: **for (*init*; *test*; *increment*) {}**
  - If *test* missing → considered true → infinite loop
  - If *increment* missing → equiv. to **while** loop
- C **for** loop design choices
  - No explicit loop variable
  - Can change anything — loop variable, test, increment — during loop
  - Can even branch into loop body! (goto label; → label: stmt;)
- C versus (e.g.) Ada:
  - C: flexible, anything goes culture; unsafe
  - Ada: prevent errors at expense of flexibility

Python **for** loop

- Format

```
for loop_var in object:
  …loop body…
[else:
  …else clause…]
```

- *object*: often a **range**
  - list of values in brackets: [1, 3, 5]
  - *range()* function: only integer arguments, optional lower bound and step size
    - range(5) ⟹ [0, 1, 2, 3, 4], range(2,7) ⟹ [2, 3, 4, 5, 6], range(0,8,2) ⟹ [0,2,4,6]
- *loop_var*: takes one of the values of range per iteration
- Else clause (optional)
  - executed when the loop terminates normally
  - **break** statement will keep it from executing:

```
for item in list:
  if item == 3:
    break
else:
  print("Didn't find item")
```

Logically-controlled loops

- Repetition depends on Boolean expression
- Simpler than count-controlled loops
- C-like **for** loop is really this
- Design issues:
  - Pre-test (**while** loop) or post-test (**until** loop)
  - Allow arbitrary exits?
  - Separate statement or special case of counting loop (e.g., C-like)

Pre-test loops

- Grammar (in general):

```
<whileStmt> → while ( <exp> ) <stmt>
```

- Semantics:

1. expression evaluated
2. if true, then <stmt> executed, goto 1
3. if false, terminate loop

- Loop body executes 0 or more times
- Can use this for all iteration

Pre-test loop operational semantics

```
loop: if (control_expression==false) goto out
        [loop  body]
        goto loop
out: …
```


Pre-test loops

- What if want loop body executed 1 or more times?
- Have to repeat loop body before loop
- Not the best way of doing things!

Post-test loops

- Test is at end of loop
- Body of loop done 1 or more times: body then test, etc.
- Called **repeat until**, **until**, or **repeat** loops
- Possible grammar rules:

  <doWhile> → do <stmt> while <exp>

  <doUntil> → do <stmt> until <expr>
- Test at end of loop; body executes at least once

Post-test loop operational semantics

- With "while"

```
loop: [loop  body]
        if (control_expression==true) goto loop
out:
```

- With "until"

```
loop: [loop body]
        if (control_expression == false) goto loop
out:
```


C **while** and **do**

- C, C++: both pre- and post-test forms
- Arithmetic control expression
- Pre-test:

  while *(exp) stmt*
- Post-test:

  do *stmt* while *(exp)*
- Java:
  - like C and C++…
  - but control expression Boolean, not arithmetic
  - cannot enter body except at beginning (no **goto** in any case)

Loops in Ada

- Allows arbitrary tests (like many languages):

```
loop
   Get(Current_Character);
   exit when Current_Character = '*';
end loop;
```

- General form: can do both pre- and post- tests, plus other
- Ada's **while** loop:

```
while Bid(N).Price < Cut_Off.Price loop
   Record_Bid(Bid(N).Price);
   N := N + 1;
end loop;
```

Loops in FORTRAN IV

```
111  FORMAT(I2,' squared=',I4)
     DO 200 I=1,20
     J = I**2
     WRITE(6,111) I,J
200  CONTINUE
```

Now, though, have **do**...**end do** loops

Loops in Lisp

- Repetition in Lisp — primarily via **recursion**
- But does have built-in loops:
  - General: (do …)
    - `(dolist (var list) {form}*)`
    - `(dotimes (var limit) {form}*)`
  - Infinite loop: `(loop {form}*)`

Loops in Lisp

- **Loop macro** — very flexible:

```
CL-USER> (loop for i from 1 to 20
               for j from 20 downto 1
               while (not (= i (+ j 1)))
               when (evenp i)
               do (format t "~s ~s~%" i j)
               collect (list i j)
               finally (print "Done!"))
2 19
4 17
6 15
8 13
10 11

"Done!"
((1 20) (2 19) (3 18) (4 17) (5 16) (6 15)
        (7 14) (8 13) (9 12) (10 11))
```


Loop macro expansion

```
(BLOCK NIL
 (LET ((I 1))
  (DECLARE (TYPE (AND REAL NUMBER) I))
  (LET ((J 20))
   (DECLARE (TYPE (AND REAL NUMBER) J))
   (SB-LOOP::WITH-LOOP-LIST-COLLECTION-HEAD (#:LOOP-LIST-HEAD-931
                                            #:LOOP-LIST-TAIL-932)
    (SB-LOOP::LOOP-BODY NIL
              (NIL NIL (WHEN (> I '20) (GO SB-LOOP::END-LOOP))
               NIL NIL NIL (WHEN (< J '1) (GO SB-LOOP::END-LOOP))
               NIL
               (UNLESS (NOT (= I (+ J 1)))
                 (GO SB-LOOP::END-LOOP)))
              ((IF (EVENP I)
                   (FORMAT T "~s ~s~%" I J))
               (SB-LOOP::LOOP-COLLECT-RPLACD
                (#:LOOP-LIST-HEAD-931 #:LOOP-LIST-TAIL-932)
                (LIST (LIST I J))))
              (NIL (SB-LOOP::LOOP-REALLY-DESETQ I (1+ I))
               (WHEN (> I '20) (GO SB-LOOP::END-LOOP)) NIL NIL
               (SB-LOOP::LOOP-REALLY-DESETQ J (1- J))
               (WHEN (< J '1) (GO SB-LOOP::END-LOOP)) NIL
               (UNLESS (NOT (= I (+ J 1)))
                 (GO SB-LOOP::END-LOOP)))
              ((PRINT "Done!")
```

Loop control and exit

- Sometimes top/bottom for loop control not sufficient
- For single (unnested) loop:
  - **break** statement (or equiv.)
  - Ada's **exit when** mechanism
- What about nested loops? How to get out of more than one loop?

Loop control

- C: provides two **goto**-like constructs
  - **break** — exit current loop/switch structure
  - **continue** — transfer control to loop test
- C/C++/Python:
  - **continue** is unlabeled
  - → skip remainder of current iteration, don't exit
- Java/Perl: labeled version of **continue**
- Ada: labeled version of **exit when:**

```
foo:
  loop
    stmts
    exit foo when condition
    stmts
   end loop foo;
```


Iteration based on data

- Control mechanism:
  - Call an iterator function that returns next element
  - Terminate when done
- **Iterator**: object with state
  - Remembers last element returned, next

```
init_iterator(it);
while (obj = it.getNextObject()) {
 process_obj(obj);
}
```


Iteration based on data

- C **for** loop —can easily be used for a user-defined iterator:

```
for (p=root; p==NULL; p = p->next){
   process_node(p);
     . . .
}
```

Python **for** statement

- **for** statement in Python — really an iterator
- Iterates over elements of a sequence or other iterable object
- Syntax:

<forStmt> → for <targetList> in <exprList> : <stmts1> [else : <stmts2>]

- <exprList> evaluated once, should → iterable object
- <stmts1> is then executed once per item provided by iterator, with the item assigned to <targetList>
- When iterator is exhausted, **else** clause is executed, if present
- If **break** occurs in <stmts1> ⟹ loop terminates without executing **else** clause
- **continue** is allowed as well

Python **for** statement

- Statements can change the <targetList> variables — next value will be assigned in same way, though
- If the sequence (e.g., a list) is modified by the loop statements:
  - Python keeps an internal counter to keep track of which item is next
  - If delete current or previous element, next item will be skipped!
  - If insert item prior to the current one, current will be processed again!
- Avoid this by making a copy of the list, e.g., with a slice:

```
for x in a[:]:
    if x < 0:
        a.remove(x)
```


Javascript object iteration

```
var o = {a:1, b:"aardvark", c:3.55};

function show_props(obj, objName) {
   var result = "";
   for (var prop in obj) {
      result += objName+"."+prop+" = "+ obj[prop] + "\n";
   }
   return result;
}

alert(show_props(o, "o"));
/* alerts :
o.a = 1
o.b = aardvark
o.c = 3.55
*/
```


Topics

- Introduction
- Selection statements
- Iterative statements
- Unconditional branching
- Guarded commands

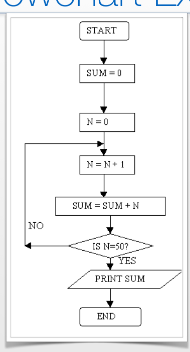
Unconditional Branching

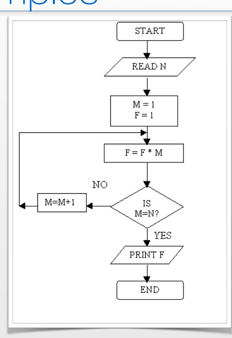
- **goto**, e.g.
- Equivalent to unconditional branch/jump in machine lang.
- Caused one of the most heated debates in 1960's and 1970's
- Major concern: readability (of "spaghetti code")
- C has goto, as you'd expect
- Some languages — don't even support **goto** statement (e.g., Java)
- C# — has **goto** statement, can be used in switch statements
- Gotos that aren't quite gotos:
  - loop exits
  - but restricted — "safer" gotos

The goto controversy

- Flowcharts: primary program design tool in 60s
- Programs often resembled flowcharts
- FORTRAN, Basic: line numbers (or labels) — branch targets
- Edsger Dijkstra (1968) → letter to the editor of *CACM*: "GoTo Considered Harmful"

Flowchart Examples





Structured programming

- Dijkstra advocated eliminating **goto** statement → conditional and iterative *structures*
- C, Pascal (& Algol)
  - developed with these structures → "structured programming revolution"
  - languages have **goto** statements, but not used much

A good use of gotos

- E.g., a natural implementation of DFSAs

```
State0:
  ch = getchar();
  if (ch =='0')
    goto State1;
  else
    goto State2;
State1:
  while ((ch = getchar()) == '0')
    ;
  Goto state5
State3:
…
```

- Difficult to see how to program this easily using purely structured programming

A rebuttal to structured

- E.C.R. Hehner (1979) — *Acta Informatica* article "do considered od: A contribution to the programming calculus"
  - Suggested that repetitive constructs weren't the best thing ever
  - argued for recursive refinement
  - claimed it was simpler and clearer

Topics

- Introduction
- Selection statements
- Iterative statements
- Unconditional branching
- Guarded commands

Guarded commands

- Dijkstra:
  - wanted loop and selection mechanisms that helped ensure correctness of programs
  - wanted to allow **nondeterminism** in programs (and avoid overcommitment)
  - ⟹ **guarded commands**
- Nondeterminism → good for concurrent programming

Guarded selection

- Form:

```
if <cond> -> <stmt>
[] <cond> -> <stmt>
[] <cond> -> <stmt>
…
fi
```

  - [] = "fatbars" — separators
  - <cond> = **guard**
  - <cond> -> <stmt> = **guarded command**

Guarded selection

- Form:

```
if <cond> -> <stmt>
[] <cond> -> <stmt>
[] <cond> -> <stmt>
…
fi
```

- Differences from standard selection:
  - guarded commands:
    - No set order
    - Any command with a true guard is eligible — nondeterminism
  - if no guard is true → exception

Guarded selection

- Example:

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

- Don't know (or care) whether a or b is max if they're equal, so why commit?
- Example:

```
if near_obstacle -> turnLeft()
[] near_obstacle -> turnRight()
[] predator_near -> speedUp()
fi
```

- In concurrent system…

Guarded iteration

- Iteration construct also guarded:

```
do <guard> -> <stmt>
   [] <guard> -> <stmt>
   [] <guard> -> <stmt>
   …
od
```

- Semantics:
  - if one or more guards is true, pick a statement and execute it
  - when all guards are false → exit loop