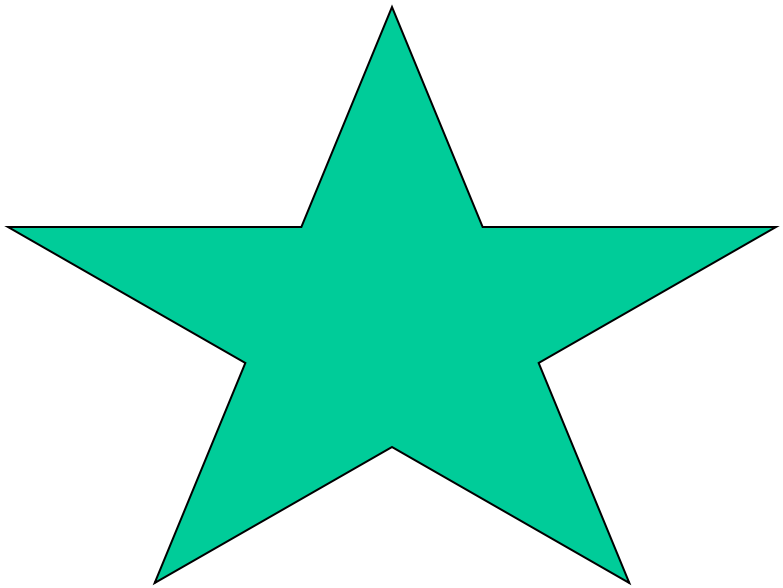
Chapter 6

Control Flow

February 14, Lecture 8



Goto Statements

- The earliest way to control the program flow
- They were partially replaced by **for**, **while**, **repeat…until**
- But kept being used in parallel for a while
- Doing other things too

- A big fight over their use and necessity
- Eventually disappeared with the advent of procedural programming

Flow Control

- Other uses of goto statements

```
    while not eof do begin
        readln(line);
        if all_blanks(line) then goto 100;
        consume_line(line)
    end;
100:
```

mid-loop exit

- **Now:** break, continue statements

Flow Control

- Other uses of goto statements

```
procedure consume_line(var line: string);
...
begin
    ...
        if line[i] = '%' then goto 100;
            (* rest of line is a comment *)
    ...
100:
end;
```

Early exit from subroutine

- **Now:** Explicit return statements with or without value

Flow Control

- Other uses of goto statements

Exit from nested subroutines

- Needs **unwinding**
- Taking out of the stack subroutines until left with outer
- Lisp can let you specify the point of exit by name (return-from)

```
function search(key : string) : string;
var rtn : string;
...
    procedure search_file(fname : string);
    ...
    begin
        ...
        for ... (* iterate over lines *)
            ...
            if found(key, line) then begin
                rtn := line;
                goto 100;
            end;
            ...
    end;
...

begin (* search *)
    ...
    for ... (* iterate over files *)
        ...
        search_file(fname);
        ...
100:    return rtn;
end;
```

Flow Control

- Other uses of goto statements
- Lexically non-nested subroutines, (dynamically nested)
- Pascal goto not enough

- Algol 60 and PL/I: Allow passing **labels as parameters** so a dynamically nested subroutine can return to a point defined by the caller

Flow Control

- Lisp and Ruby provide **throw…catch** mechanisms
- The throw specifies a tag which then appears in the catch

```
def searchFile(fname, pattern)
    file = File.open(fname)
    file.each {|line|
        throw :found, line if line =~ /#{pattern}/
    }
end

match = catch :found do
    searchFile("f1", key)
    searchFile("f2", key)
    searchFile("f3", key)
    "not found\n"           # default value for catch,
end                         # if control gets this far
print match
```

Flow Control

- Lisp and Ruby provide **throw…catch** mechanisms
- The throw specifies a tag which then appears in the catch

```
def searchFile(fname, pattern)
    file = File.open(fname)
    file.each {|line|
        throw :found, line if line =~ /#{pattern}/
    }
end

match = catch :found do
    searchFile("f1", key)
    searchFile("f2", key)
    searchFile("f3", key)
    "not found\n"           # default value for catch,
end                         # if control gets this far
print match
```

Flow Control

- **Exceptions** are used to exit when something goes wrong and a callee subroutine cannot proceed. Program `backs-out' to somewhere where this can be handled

- Here is a way to do it: return status from calls

```
status := my_proc(args);
if status = ok then ...
```

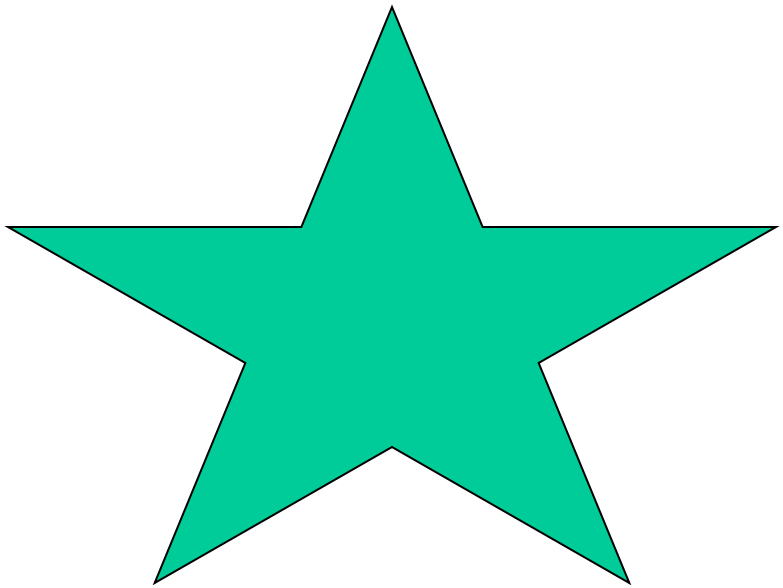
- Some languages provide exception handlers
- Will see more later

Flow Control

- **Continuations** are abstractions that capture a context in which execution can continue
- At low-level they are pointers to code along with referencing environments

- They are first-class values in some languages like Scheme
- It allows the definition of control-flow constructs (subsumes most of the usual ones)

- Continuations are important but can be abused



Sequencing

- Controls the sequence with which side effects occur
  - Second instruction after the first
  - **Compound statements** (begin…end) where single statements expected
  - A compound statement preceded by declaration is also called a **block**

- Value of statement/expression
  - Algol and C it's the value of the final element
  - Lisp allows last, first, or second
  - Sequencing in Lisp is used when deviating from functional mode

- Side-Effect Freedom for subroutines (Euclid, Turing)
  - Ensures **idempotence**, same result every time called

Sequencing

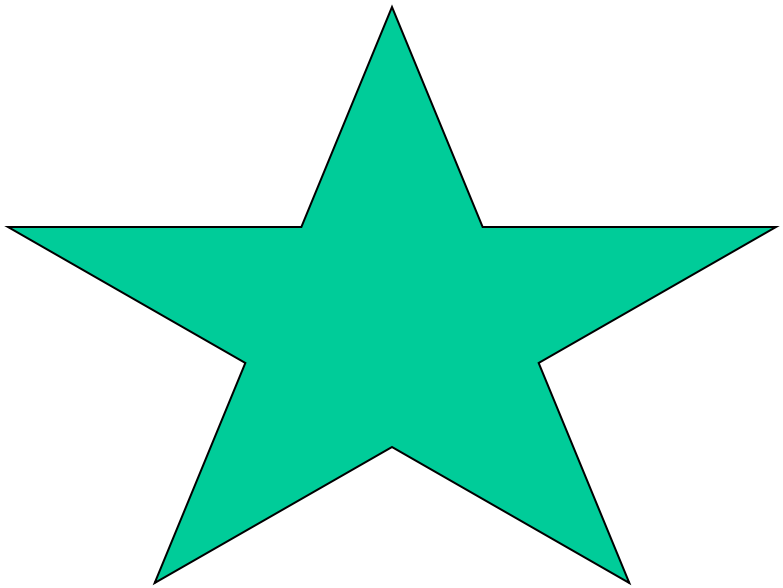
- Side-effect may be **desired** for subroutines
- Example is random number generator
- rand() returns a different number each time called

```
procedure srand(seed : integer)
    -- Initialize internal tables.
    -- The pseudo-random generator will return a different
    -- sequence of values for each different value of seed.

function rand() : integer
    -- No arguments; returns a new "random" number.
```

```
procedure rand(var n : integer)
```

recast



Selection

- Introduced in Algol 60

```
if condition then statement
else if condition then statement
else if condition then statement
...
else statement
```

- Ambiguity
- To avoid ambiguity statement is not allowed to start with if (Algol)
- Associate an else with the closer unmatched then (Pascal)

Selection

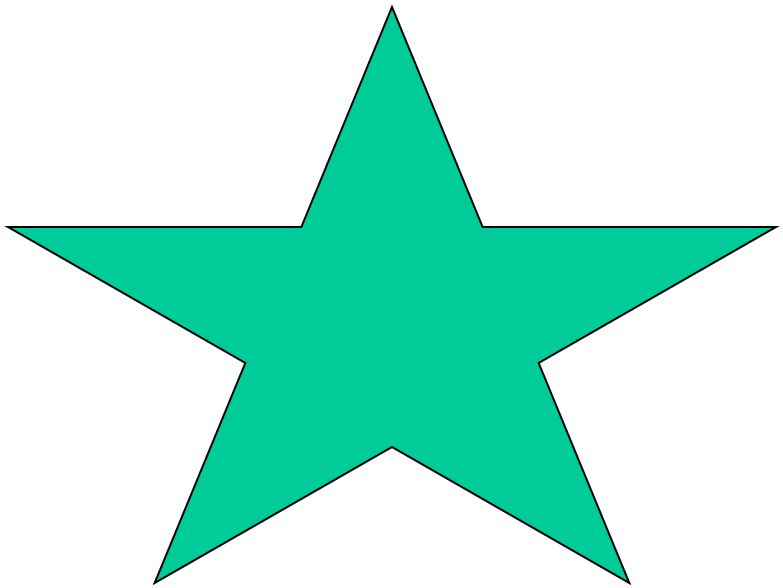
- Slightly better to read

```
IF a = b THEN ...
ELSIF a = c THEN ...
ELSIF a = d THEN ...
ELSE ...
END
```

Modula-2

```
(cond
    ((= A B)
        (...))
    ((= A C)
        (...))
    ((= A D)
        (...))
    (T
        (...)))
```

Lisp



Short-circuit conditions

- Most machines have special instructions for conditional jump
- So the purpose of the **if** is not to save a Boolean variable somewhere but to cause branch control to other locations
- This allows the generation of very efficient code

Short-circuit conditions

- Example in Pascal (no short-circuit allwowed)

```
if ((A > B) and (C > D)) or (E ≠ F) then
        then_clause
else
        else_clause
```

```
        r1 := A              -- load
        r2 := B
        r1 := r1 > r2
        r2 := C
        r3 := D
        r2 := r2 > r3
        r1 := r1 & r2
        r2 := E
        r3 := F
        r2 := r2 ≠ r3
        r1 := r1 | r2
        if r1 = 0 goto L2
L1:     then_clause          -- (label not actually used)
        goto L3
L2:     else_clause
L3:
```

Short-circuit conditions

- Example in C (using jump code)

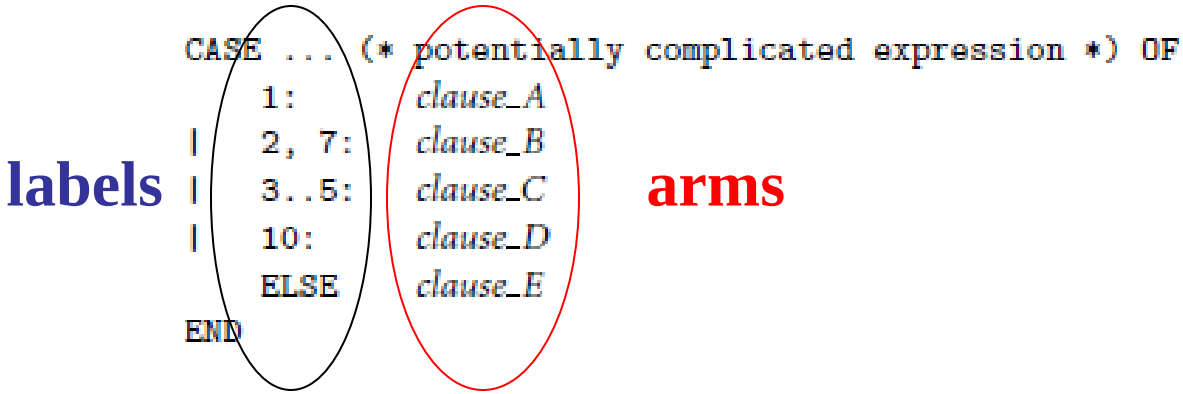
```
if ((A > B) and (C > D)) or (E ≠ F) then
        then_clause
else
        else_clause
```

```
        r1 := A
        r2 := B
        if r1 <= r2 goto L4
        r1 := C
        r2 := D
        if r1 > r2 goto L1
L4:     r1 := E
        r2 := F
        if r1 = r2 goto L2
L1:     then_clause
        goto L3
L2:     else_clause
L3:
```

Short-circuit conditions

- Case-Switch statements

```
i := ... (* potentially complicated expression *)
IF i = 1 THEN
     clause_A
ELSIF i IN 2, 7 THEN
     clause_B
ELSIF i IN 3..5 THEN
     clause_C
ELSIF (i = 10) THEN
     clause_D
ELSE
     clause_E
END
```



equivalent

- Labels can be any type that consists of discrete values
  - Including strings in C#

Short-circuit conditions

- Case-Switch statements
- The main motivation is not syntactic use but rather **efficiency**

```
    r1 := ...              -- calculate tested expression
    if r1 ≠ 1 goto L1
    clause_A
    goto L6
L1: if r1 = 2 goto L2
    if r1 ≠ 7 goto L3
L2: clause_B
    goto L6

L3: if r1 < 3 goto L4
    if r1 > 5 goto L4
    clause_C
    goto L6
L4: if r1 ≠ 10 goto L5
    clause_D
    goto L6
L5: clause_E
L6:
```

Direct translation

```
    goto L6            -- jump to code to compute address
L1: clause_A
    goto L7
L2: clause_B
    goto L7
L3: clause_C
    goto L7
    ...
L4: clause_D
    goto L7
L5: clause_E
    goto L7

L6: r1 := ...          -- computed target of branch
    goto *r1
L7:
```

address to jump

Jump tables

- General form

```
T:  &L1          -- tested expression = 1
    &L2
    &L3
    &L3
    &L3
    &L5
    &L2
    &L5
    &L5
    &L4          -- tested expression = 10
```

```
L6: r1 := ...            -- calculate tested expression
    if r1 < 1 goto L5
    if r1 > 10 goto L5       -- L5 is the "else" arm
    r1 := r1 - 1             -- subtract off lower bound
    r2 := T[r1]
    goto *r2
L7:
```

Jump tables

- Case statements may be bad if value range is non-dense
- Alternative ways
  - Sequential testing (O(n), ok when small)
  - Binary search
  - Hash table (attractive when the range of values is big but there are many missing values)
- Sophisticated compilers can do the right thing

Jump tables

- Syntactic differences between languages
- C is somewhat unusual

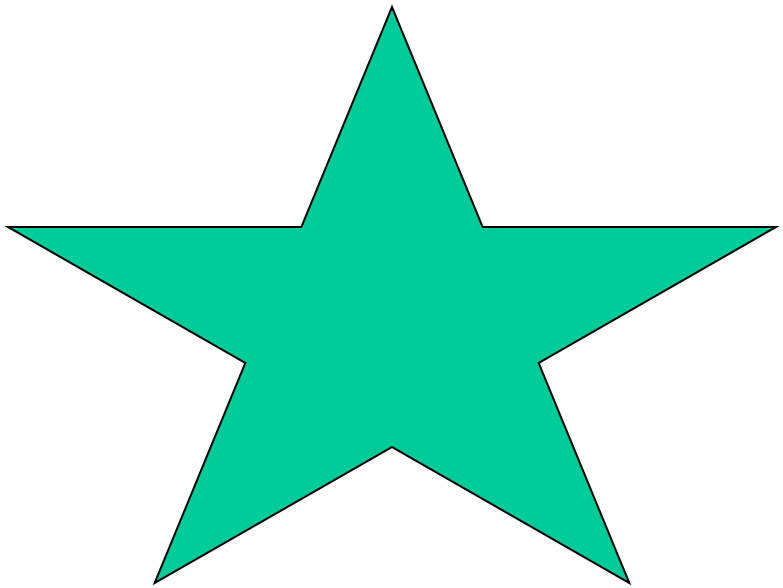
```
switch (... /* tested expression */) {
    case 1:  clause_A
             break;
    case 2:
    case 7:  clause_B
             break;
    case 3:

    case 4:
    case 5:  clause_C
             break;
    case 10: clause_D
             break;
    default: clause_E
             break;
}
```

No range allowed

```
letter_case = lower;
switch (c) {
    ...
    case 'A' :
        letter_case = upper;
        /* FALL THROUGH! */
    case 'a' :
        ...
        break;
    ...
}
```

fall-through



Loops and Recursion

- Used to execute same thing repeatedly
- Functional uses mostly recursion
- Imperative uses mostly loops,
  - As with statements they are used for their side-effects

- Enumeration-controlled loop: values over a finite set
- Logically-controlled loop: based on monitoring boolean conditions