

---

**CSc 520**

**Principles of Programming  
Languages**

***26 : Control Structures — Introduction***

Christian Collberg

[collberg+520@gmail.com](mailto:collberg+520@gmail.com)

Department of Computer Science

University of Arizona

Copyright © 2008 Christian Collberg

# Control Flow

- We need some way of **ordering** computations:
- **sequencing**
- **selection**
- **iteration**
- **procedural abstraction** — being able to treat a collection of other control constructs as a single unit, a subroutine.
- **recursion**
- **concurrency**
- **nondeterminacy** — being able to explicitly state that the ordering between two statements is unspecified, and, possibly should be selected randomly/fairly.

# Control Flow — Paradigms

---

- **Functional languages** — recursion and selection are important, iteration and sequencing not.
- **Procedural languages** — iteration, sequencing, selection are important, recursion not.
- **Logic languages** — the programmer gives rules that restrict control flow, the interpreter deduces an execution ordering that satisfies these rules.

---

# Operators

# Prefix, Infix, Postfix

- Languages use prefix, infix, or postfix notation for operators in expressions.
- This means that the operator comes before, among, or after its operands.
- Lisp/Scheme uses **Cambridge Polish** notation (a variant of prefix):  
$$(* (+ 5 6) 7)$$
- Postscript and Forth use postfix notation.
- Smalltalk uses infix notation.

# Smalltalk — Binary Messages

- A **binary** message  $M$  to receiver  $R$  with argument  $A$  has the syntax

$R \ M \ A$

- For example:

$8 \ + \ 9$

This sends the message  $+$  to the object 8 with the argument 9.

# Smalltalk — Keyword Messages

- A **keyword** message  $M$  to receiver  $R$  with arguments  $A_1, A_2, A_3, \dots$  has the syntax

$R \ M_1: \ A_1 \ M_2: \ A_2 \ M_3: \ A_3 \ \dots$

- For example:

`DeannaTroi kiss: cheek how: tenderly`

This sends the message `kiss:how:` to the object `DeannaTroi` with the arguments `cheek` and `tenderly`. In Java we would have written:

`DeannaTroi.kisshow(cheek,tenderly)`

# Operator Precedence

- The **precedence** of an operator is a measure of its **binding power**, i.e. how strongly it attracts its operands.
- Usually  $*$  has higher precedence than  $+$ :

$$4 + 5 * 3$$

means

$$4 + (5 * 3),$$

not

$$(4 + 5) * 3.$$

- We say that  $*$  binds harder than  $+$ .



# Operator Associativity

- The **associativity** of an operator describes how operators of equal precedence are grouped.
- $+$  and  $-$  are usually **left associative**:

$$4 - 2 + 3$$

means

$$(4 - 2) + 3 = 5,$$

not

$$4 - (2 + 3) = -1.$$

We say that  $+$  **associates to the left**.

- $^$  associates to the right:

$$2^3^4 = 2^{(3^4)}.$$

# Case Study — C

---

- C has so many rules for precedence and associativity that most programmers don't know them all.
- See the table on the next slide.

# Case Study — C...

OPERATOR	KIND	PREC	ASSOC
a[k]	Primary	16	
f(⋯)	Primary	16	
.	Primary	16	
->	Primary	16	
a++, a--	Postfix	15	
++a, --a	Unary	14	
~	Unary	14	
!	Unary	14	
-	Unary	14	
&	Unary	14	
*	Unary	14	

OPERATOR	KIND	PREC	ASSOC
*, /, %	Binary	13	Left
+, -	Binary	12	Left
<<, >>	Binary	11	Left
<, >, <=, >=	Binary	10	Left
==, !=	Binary	9	Left
&	Binary	8	Left
^	Binary	7	Left
	Binary	6	Left
&&	Binary	5	Left
	Binary	4	Left
? :	Ternary	3	Right
=, +=, -=, *=, / =, % =, << =, >> =, & =, ^ =,   =	Binary	2	Right
,	Binary	1	Left

---

# Variables

# Value vs. Reference Model

- **l-value** — an expression that denotes a location, such as the left-hand side in  $x := \dots$ ,  $x[i] := \dots$ ,  $x.a[i] \rightarrow v := \dots$ .
- **r-value** — an expression that denotes a value, such as the right-hand side in  $\dots := x$ ,  $\dots := x[i]$ ,  $\dots := x.a[i] \rightarrow v$ ,  $\dots := 3 + x$ .
- Pascal, C, Ada use a **value model** of variables. In  $\dots := x$ ,  $x$  refers to the value stored in  $x$ .
- Clu (and other languages) use a **reference model** for variables. In  $\dots := x$ ,  $x$  is a reference to the value stored in  $x$ .

# Value vs. Reference Model...

- In Pascal, after the statements

```
b := 2;
```

```
c := b;
```

both `b` and `c` would hold the value 2. In Clu, `b` and `c` would both point to the same object, which contains the value 2.

- Java uses a value model for `int`, `float`, etc, but a reference model for `String`. Hence

```
int i,j;
```

```
String s,t;
```

```
if (i==j) ...
```

```
if (s==t) ...
```

can be confusing for novel programmers.

---

# Expressions

# Order of Evaluation

- Many languages allow the compiler to reorder operations in an expression, for efficiency.
- Java requires strict left-to-right evaluation. Why?
- If the expression (`b, c, d` are 32-bit `ints`)

`b - c + d`

is reordered as

`b + d - c`

then an overflow can occur if `b + d` doesn't fit in an `int`.



# Order of Evaluation...

- Let  $a, b, c$  be 32-bit floats, where  $a$  is small,  $b, c$  are large, and  $b = -c$ .

- Then the expression

$$(a+b)+c$$

might evaluate to 0 (due to a loss of information), while

$$a+(b+c)$$

would evaluate to  $a$ .

# Case Study — Pascal

- Pascal does *not* use **short-circuit evaluation**. Hence, this makes for problems:

`if (x<>0) and (y/x > 5) then`

- Pascal has non-intuitive precedence:

`4 > 8 or 11 < 3`

is parsed as

`4 > (8 or 11) < 3`

Hence, it becomes necessary to insert parenthesis.

---

# Control-Flow Statements

# Statement vs. Expression Orientation

- In Pascal, Ada, Modula-2, `if`, `while`, etc. are **statements**. This means that they are executed for their side-effects only, and return no value.
- In Algol68 `if`, `while`, etc. are **expressions**, they can have both side-effects and return values:

```
begin
  x := if b<c then d else e;
  y := begin f(b); g(c) end;
  z := while b<c do g(c) end;
  2+3
end
```

This compound block returns 5.

# Unstructured Control-Flow

- In the early days of FORTRAN, there were no structured control-flow statements (these were introduced in Algol 60).
- Instead, programmers built up structured `ifs`, `whiles`, etc, using `gotos`:

```
        IF a .LT. B GOTO 10
           ...
        GOTO 20
10:           ...
20:
```

This is an `if-then-else`-statement.

# Case Study — Pascal: goto

- Pascal has no exception handling mechanism. Gotos were the only way of, say, jumping to the end of the program on an unrecoverable error.
- Labels have to be integers and have to be declared.

```
goto label;
```

```
...
```

```
label:
```

```
procedure P ( );
```

```
    label 999;
```

```
    ...
```

```
    goto 999;
```

```
    ...
```

```
999:
```

```
end;
```

---

# Statements — Selection

# Case Study — Pascal: if

---

```
if boolean expression then
    statement
else
    if boolean expression then
        statement
    else
        begin
            statement
            statement
            statement
        end
    end
```

- The **else** is always matched with the closest nested if.



# Case Study — Modula-2: if

- The **ELSIF** part of an **IF**-statement in Modula-2 is a convenient addition from Pascal:

```
IF boolean expression THEN  
    statement-sequence  
ELSIF boolean expression THEN  
    statement-sequence  
ELSIF boolean expression THEN  
    statement-sequence  
ELSE  
    statement-sequence  
END
```

# Case Study — Pascal: case

```
case ordinal expression of
  list of cases: statement;
  list of cases: statement;
  list of cases: statement;
otherwise statement
end;
```

- **otherwise** is optional.
- The *list of cases* looks like this: **1,2,7..9**. I.e. it can contain ranges.
- case-statements can be implemented as nested `ifs`, jump-tables (most common), or hash-tables, depending on what is most efficient.

# Case Study — C: case

- In 1990 AT&T's long distance service fails for nine hours due to a wrong `break` statement in a C program.

```
switch (e) {  
    0 :  
    1 :   $S_1$ ;  
        break;  
    2 :   $S_2$ ;  
    3 :   $S_3$ ;  
        break;  
}
```

⇐ Really meant to fall-through here?!?!

- C's design allows several cases to share the same statement (as 0 and 1 do above).

# Case Study — FORTRAN: goto

- In FORTRAN, you can simulate a case statement using **computed gotos**:

```
                GOTO (15, 20, 30) I
15:             ...
20:             ...
30:             ...
```

If  $I=1$ , we'll jump to 15; if  $I=2$ , we'll jump to 20; if it's 3, we'll jump to 30, otherwise we'll do nothing.

---

# Statements — Iteration

# Case Study — Pascal: for

---

```
for index := start to stop do  
    statement;  
for index := start downto stop do  
    statement;
```

- The index must be declared outside the loop.
- Only ordinal datatypes are allowed.
- You can only increment the index variable with  $\pm 1$ !

# Case Study — Modula-2: FOR

- Modula-2 generalizes Pascal's for-loop, so that it's possible to iterate by an arbitrary amount:

```
( * The BY-part is optional.  
  step must be a constant. * )  
FOR i := from TO to [BY step] DO  
  statement-sequence  
END
```

- *step* still has to be constant, though!

# Case Study — Modula-3: FOR

- Modula-3, finally, provides a FOR-loop in its full generality:

```
FOR id := first TO last BY step DO
    S
END
```

- `id` is a read-only variable with the same type as `first` and `last`.
- `first`, `last` and `step` are executed once.
- `step` can be a run-time expression, not just a constant. (At least, I think so — Scott says otherwise, and the manual is silent. Anyone care to check what the compiler thinks?)



# Case Study — Modula-3: FOR

```
FOR id := first TO last BY step DO  
    S  
END
```

- If `step` is negative, the loop iterates downwards.
- It is non-trivial to implement a fully general FOR-loop. See the next slide for how Modula-3's FOR-statement is translated.
- The index variable `id` is automatically defined by the loop.
- In Pascal/Modula-2, the programmer had to define it herself outside the loop. This lead to the question **what value will `id` have after the end of the loop?** Either the compiler got it wrong, or the programmer got it wrong.

# Case Study — Modula-3: FOR...

---

```
FOR id := first TO last BY step DO S END
```

⇓ ⇓ ⇓

```
VAR i := ORD(first); done := ORD(last); delta := step;
BEGIN
  IF delta >= 0 THEN
    WHILE i <= done DO
      WITH id=VAL(i,T) DO S END; INC(i,delta);
    END
  ELSE
    WHILE i >= done DO
      WITH id=VAL(i,T) DO S END; INC(i,delta);
    END
  END END END
```

# Case Study — Pascal: loops

---

```
while boolean expression do  
    statement;
```

```
repeat  
    statement;  
    statement;  
until boolean expression;
```

- Note the asymmetry: the **while** statement body can only contain one statement.

# Case Study — Modula-2: loops

- Modula-2 adds an infinite loop:

**LOOP**

*statement-seq* ( \* **EXIT** can occur here. \* )

**END**

- This makes it convenient to exit a loop in the middle:

LOOP

. . . .

IF . . . THEN EXIT;

. . . .

END

# Case Study — Algol 60

- Algol 60 has **one** loop construct:

`for ::= for id := list do stat`

`list ::= enum { , enum }`

`enum ::= expr |  
          expr step expr until expr |  
          expr while condition`

- id takes on values specified by a sequence of enumerators.
- Each expression is re-evaluated at the top of the loop.

# Case Study — Algol 60...

---

- Each of the following is equivalent:

`for i := 1, 2, 5, 7, 9 do ...`

`for i := 1 step 2 until 10 do ...`

`for i := i, i + 2 while i < 10 do ...`

- This generality is usually overkill...

---

# Recursion

# Tail Recursion

---

- A function is **tail-recursive** if there is no more work to be done after the recursive call.
- Tail-recursive functions are important because they can be easily be made iterative — no stack space needs to be allocated dynamically.
- For tail-recursive functions the compiler can **reuse** the space of the current stack frame instead of allocating a new one for the recursive call.



# Tail Recursion...

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



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

# Tail Recursion...

---

- You can often transform a non-tail-recursive function into a tail-recursive one.
- The idea is to pass a **continuation** of the work that is to be done **after** the call as a parameter to the call.
- This is called **continuation-passing style** (CPS).
- The next slide shows how the factorial function has been made tail-recursive using the CPS transformation.

# Tail Recursion...

---

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

```
(define (fact-cps n C)
  (if (= n 1)
      (C 1)
      (fact-cps (- n 1) (
        lambda(v) (C (* n v))))))
```

```
(fact-cps 5 (lambda(v) (display v)))
```

# Readings and References

---

- Read Scott, pp. 233–242, 249–257, 260–278, 284–291