Control

Contents

Simple statements

(in imperative languages)

assignment, empty statement,

procedure call, jump, exit,

compound statement (yes, in context!).

Structured statements

There are three fundamental and indispensable mechanisms for arranging simple statements:

- sequence (**`begin-end`**) — compound statement
- selection (**`if-then-else`**) — conditional statement
- iteration (**`while-do`**) — loop statement

They are used to build structured statements.

All other structuring mechanisms can be easily derived from the basic three mechanisms:

`if` $\mathcal{C}$ `then` $\mathcal{S}$ $\equiv$ `if` $\mathcal{C}$ `then` $\mathcal{S}$ `else null`

`repeat` $\mathcal{S}$ `until` $\mathcal{C}$ $\equiv$ $\mathcal{S}$`;` `while` $\neg\mathcal{C}$ `do` $\mathcal{S}$

`for` i `:=` lo `to` hi `do` $\mathcal{S}$ $\equiv$
  i `:=` lo`;` `while` $i \le hi$ `do`
    `begin` $\mathcal{S}$`;` i `:=` $succ(i)$ `end`

`case` i `of`
  C_1`:` $\mathcal{S}_1$`;` `......` $\equiv$
  `if` i `=` C_1 `then` $\mathcal{S}_1$ `else ......`

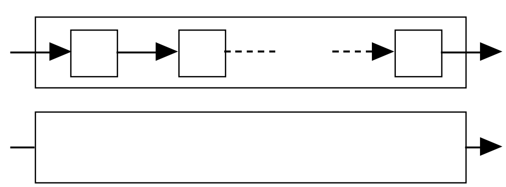
---

and so on.

Sequence

- Algol, Pascal, Ada, ... `begin ... end`
  (there also are blocks)
- C `{ ... }`
- Fortran (older) nothing
- Prolog implicit
  `a :- b, c, d.`
  Evaluate (execute) `b`, then `c`, then `d`.

A compound statement **`begin ... end`** is treated syntactically as a simple statement. This is an important abstraction principle.



The inner structure can be "abstracted away".

`begin` and `end` are syntactic brackets. In Algol 68 there are pairs `if-fi`, `do-od`, `case-esac` etc.

Selection

`if` C `then` $S1$ `else` $S2$ Pascal

A possible ambiguity of
`if` $C1$ `then if` $C2$ `then` $S1$ `else` $S2$
is resolved in Pascal by the nearest-then rule:
`if` $C1$ **`then begin if`** $C2$ **`then`** $S1$ **`else`** $S2$ **`end`**

`if` C `then` *single-$S1$* `else` $S2$ Algol 60

No ambiguity is possible:
`if` $C1$ `then if` $C2$ `then` $S1$ `else` $S2$
would be syntactically incorrect.

`if` C `then` $S1$ `else` $S2$ `end` Modula-2

No ambiguity—one of these must be used:
`if` $C1$ **`then if`** $C2$ **`then`** $S1$ **`else`** $S2$ **`end end`**
`if` $C1$ **`then if`** $C2$ **`then`** $S1$ **`end else`** $S2$ **`end`**

`if` C `then` $S1$ `else` $S2$ `end if` Ada

Nested selection in Ada:
`if` C_1 `then` S_1 `elsif` C_2 `then` S_2
`......` `elsif` C_n `then` S_n
`else` S_{n+1} `end if`

Special forms of selection

Computed goto in Fortran: primitive.

```
GO TO (label1, ..., labeln), expression
```

Assigned goto in Fortran: peculiar.

```
ASSIGN labeli TO variable
GO TO variable(label1, ..., labeln)
```

Switch statement in C: similar to computed goto.

```
switch(expression){
   case const1: S1;
   ...
   case constn: Sn;
   default: Sn+1;}
```

After S_i has been executed, control "falls through" to the subsequent case: S_{i+1} is executed next. This can be avoided by adding `break` statements:

```
switch(expression){
   case const1: S1; break;
   ...
   case constn: Sn; break;
   default: Sn+1;}
```


Case statement in Pascal: cases are separate.

```
case expression of
   constList1: S1;
   ...
   constListn: Sn;
   else Sn+1
end
```

Case statement in Ada is similar.

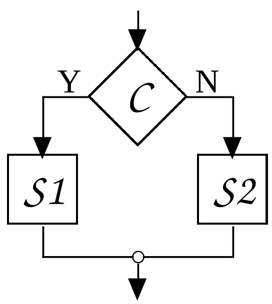
```
case expression is
   when constList1 => S1;
   ...
   when constListn => Sn;
   when others => Sn+1;
end case;
```

Selection in Prolog is driven by success-failure, not by truth-falsity. It is implicit in backtracking.

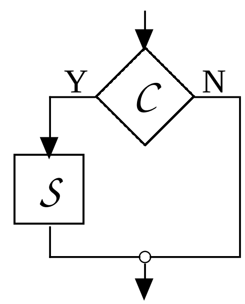
```
eat_or_drink(Stuff) :-
   solid(Stuff), eat(Stuff).

eat_or_drink(Stuff) :-
   liquid(Stuff), drink(Stuff).
```

Graphical representation of selection: flowgraphs (flow diagrams, flowcharts)

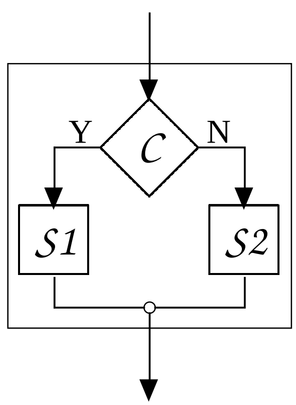
`if-then-else`

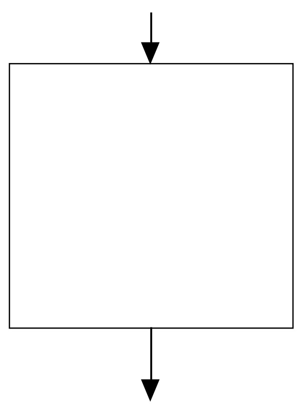


`if-then`

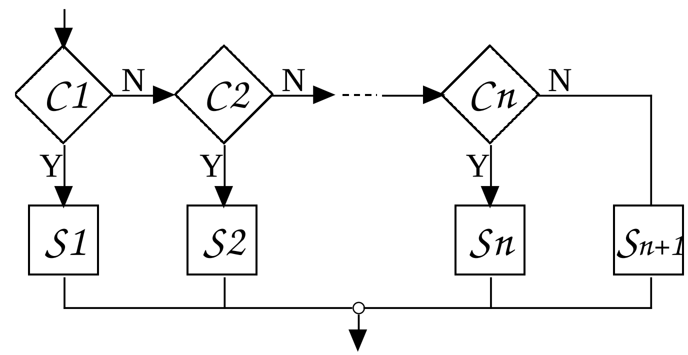


The abstraction principle:
`begin if` *C* `then` *S1* `else` *S2* `end`
is a simple statement.

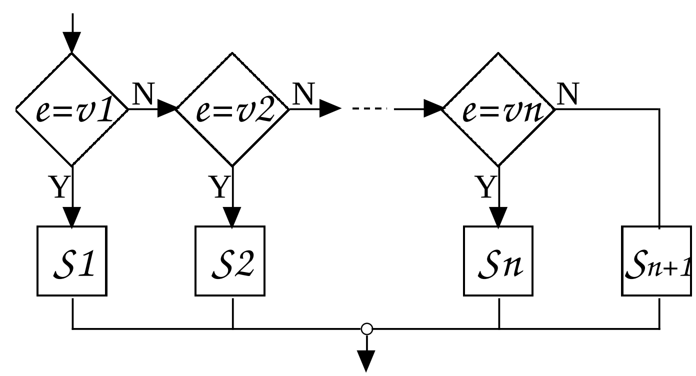




`if-then-elsif-then-...-else`



`case` *e* `of` *v1*`:` *S1*`; ... else` *Sn+1* `end`



Iteration

Variations: pretest iteration or posttest iteration.

`while` $\mathcal{C}$ `do` $\mathcal{S}$ — Pascal

`repeat` $\mathcal{S}$ `until` $\mathcal{C}$

`while (`*expr*`)` $\mathcal{S}$`;` — C

`do` $\mathcal{S}$ `while (`*expr*`);`

`while` $\mathcal{C}$ `loop` $\mathcal{S}$ `end loop;` — Ada

no posttest iteration

In Ada, the prefix `while` $\mathcal{C}$ is an extension of the basic iterative statement:

`loop` $\mathcal{S}$ `end loop;`

Another prefix: `for` *i* `in` *range*

The bare `loop` statement must be stopped from inside the loop.

Forced exit closes the nearest iteration:

```
exit;    -- no conditions
exit when C;
```


The `while` prefix is an abbreviation:

`while` $\mathcal{C}$ `loop` $\mathcal{S}$ `end loop;`

is equivalent to

```
loop
  exit when not C;
  S
end loop;
```

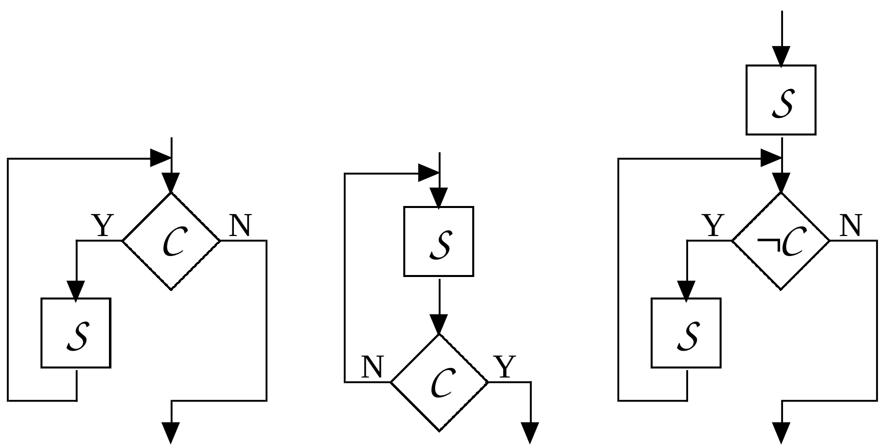
☒ Example of use of `exit`: sum up non-zero data.

`SUM := 0;`
 	
`loop`
` get(X);`
` exit when X = 0;`
` SUM := SUM + X;`
`end loop;`
`put(SUM);`
	`SUM := 0;`
	`get(X);`
	`while X /= 0 loop`
	` SUM := SUM + X;`
	` get(X);`
	
	`end loop;`
	`put(SUM);`

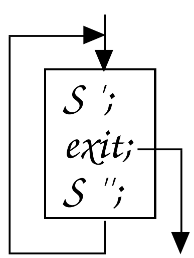
Simpler, more intuitive.	Condition reversed, `get` appears twice.

Graphical representation of iteration

`while-do` `repeat-until` `repeat-until`



`loop-end loop`



For-loops ("counter-controlled") are historically earlier and less general than condition-controlled iterative structures.

DO *label* *var* = *lo*, *hi* — Fortran
...
label CONTINUE

DO *label* *var* = *lo*, *hi*, *incr*

---

for *var* := *expr* do *S* — Algol 60

for *var* := *lo* step *incr* until *hi* do *S*

for *var* := *expr* while *C* do *S*

Iterators can be combined:

for *i* := *0*, *i+1* while $i \le n$ do *S(i)*

---

for *var* := *lo* to *hi* do *S* — Pascal

---

for *var* in *range* — Ada
loop *S* end loop;

for *var* in reverse *range*
loop *S* end loop;

`for (`*expr1*`;` *expr2*`;` *expr3*`)` *S*`;` C

This is equivalent to the following:

expr1`;`
`while (`*expr2*`) {` *S*`;` *expr3*`; }`

A typical application: *expr1* initializes a variable, *expr2* uses it and *expr3* increments it.

```
for (i = 0; i <= n ; i++) S(i);
```

Default *expr2* to "true". This is "**`loop`** *S* **`end loop`** ":

```
for (;;) S;
```

Iteration in Prolog is expressed by recursion. The same, by the way, can be done in Pascal etc.:

```
procedure iter(C: boolean; S: procedure) is
begin if C then S; iter(C, S); end if; end;
```

One Prolog example:

```
printlist([E | Es]) :-
  write(E), printlist(Es).
printlist([]) :-
  nl.
```

(The same effect is achieved if the order of clauses is inverted.)

Jump (the `goto` statement)

Unconstrained transfer of control is the only mechanism available in low-level languages. One-way selection and `goto` are all we need to express every other control structure.

The mechanism is dangerous, hurt readability, and should be avoided—advanced control structures work well for all typical and less typical uses.

Some languages restrict `goto` (e.g. do not allow jumps into an iteration or selection) and make it a pain to use it. Ada requires each label to be visible from far away, for all managers to see ☺:

```
      SUM := 0;
      loop
         get(X);
         if X = 0 then goto DONE; end if;
         SUM := SUM + X;
      end loop;
   <<DONE>>
      put(SUM);
```

`goto` may leave “unfinished business”—active control structures that must be "folded" at once.

Guarded commands

A very general form of selection is Dijkstra's `if` with guarded commands (a guard is a boolean expression):

```
if
   guard₁ → statements₁          □
   guard₂ → statements₂          □
   ...                           □
   guardₖ → statementsₖ
fi
```

Evaluate all guards in parallel, choose any true guard, execute its statements. If more than one guard is true, the choice is non-deterministic.

It is an execution *error* not to find any true guards.

☒ This has "overlapping" guards:

```
if X ≤ Y →  Max := Y
□  X ≥ Y →  Max := X
fi
```


A very general form of iteration is Dijkstra's `do` with guarded commands (again, boxes □ are used to separate guard-statements pairs):

```
do
   guard₁ → statements₁          □
   guard₂ → statements₂          □
   ...                           □
   guardₘ → statementsₘ
od
```

Repeat this: evaluate all guards in parallel, choose any true guard (perhaps non-deterministically), execute its statements. Terminate the loop if none of the guards is true.

☒ X, Y are integer variables with non-negative values. The loop terminates when X = Y.

```
do  X ≠ Y  →
   if X < Y  →  Y := Y - X □
      X > Y  →  X := X - Y
   fi
od
```

☒ (See the textbook.) When does this loop terminate? [Note simultaneous assignments.]

```
q1, q2, q3, q4 := Q1, Q2, Q3, Q4;
do
   q1 > q2  →  q1, q2 := q2, q1
   ▯
   q2 > q3  →  q2, q3 := q3, q2
   ▯
   q3 > q4  →  q3, q4 := q4, q3
od
```

Finally: how can we express `if-then-else`, `if-then`, `case-of`, `while-do`, `repeat-until` using guarded commands?

Summary

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....

....