

Repetition (= Iteration = Loop) Constructs

Two kinds of loop:

- *definite*: iteration count is known before first iteration
- *indefinite*: termination condition is computed in loop's body

The distinction is a historical artifact: Definite iteration was easy to map efficiently onto certain early processors' architectures and instruction sets (decrement-and-test).

Indefinite iteration

- Test *precedes* body (\Rightarrow iteration count ≥ 0)

Pascal:

while B **do** S

Pascal equivalent:

while B **do** S

=

if B **then begin** S ; **while** B **do** S **end**

C equivalent:

`while(B) S`

GCN equivalent:

do $B \rightarrow S$ **od**

C for-loop:

`for(S_0 ; B ; S_1) S_2`

=

`S_0 ; while(B){ S_2 ; S_1 }`

- Test *follows* body (\Rightarrow iteration count ≥ 1)

Pascal:

repeat S **until** B (*indefinite iteration*)

Pascal equivalent:

[**repeat** S **until** B]

=

[S ; **while not** B **do** S]

C equivalent:

`do S while(!B)`

GCN equivalent:

S ; **do** $\neg B \rightarrow S$ **od**

Definite iteration

- ascending

Pascal (v : integer)

for $v := E_0$ **to** E_1 **do** S (*can't assign to v in S *)

compare with ($\forall v: E_0 \leq v \leq E_1: S$)

Pascal equivalent:

$v := E_0$;

while $v \leq E_1$ **do begin** S ; $v := v + 1$ **end**;

$v := \perp$

C equivalent (indefinite):

`for(int $v = E_0$; $v \leq E_1$; $v++$) S ; $v = \perp$`

GCN equivalent (indefinite):

$v := E_0$;

do $v \leq E_1 \rightarrow S$; $v := v + 1$ **od**;

$v := \perp$

- for step size $\neq 1$

Modula-2:

FOR $v := E_0$ **TO** E_1 **BY** C **DO** S **END**

GCN equivalent:

$v := E_0$;

do

$(C \geq 0 \wedge v \leq E_1) \vee (C \leq 0 \wedge v \geq E_1)$

$\rightarrow S$; $v := v + C$

od

- for step size = -1

Pascal

for $v := E_0$ **downto** E_1 **do** S

Modula-2 equivalent:

FOR $v := E_0$ **TO** E_1 **BY** -1 **DO** S **END**; $v := \perp$

Pascal equivalent:

$v := E_0$;

while $v \geq E_1$ **do begin** S ; $v := v - 1$ **end**;

$v := \perp$

Guardless loops

- Modula-2

LOOP

S_0 ;

IF E **THEN** **EXIT** **END**;

S_1

END

C:

`for(;;)`

{

S_0 ;

`if(E) break;`

S_1

}

GCN:

S_0 ; **do** $\neg E \rightarrow S_1$; S_0 **od**

Reasoning about loops

while E **do** S { $\neg E$ }

Loop's termination \Rightarrow its guard is false.

Modula-2 allows EXIT only in (guardless) LOOP.

C, however, allows

`while(E){ ... break; ... } { $\neg E$?`

Using break in guarded loops *breaks* the abstraction.

Syntactic issues: the Semicolon Controversies

- Should the semicolon be a statement *separator* or a statement *terminator*?
- Should the *empty statement* be permitted?

A (partial) statement grammar (semicolon is a separator, empty statement not permitted):

$$S ::= \mathbf{stmt} \mid \mathbf{begin} \ SL \ \mathbf{end} \qquad G_0$$

$$\mid \mathbf{while} \ \mathit{expr} \ \mathbf{do} \ S$$

$$SL ::= SL ; S \mid S$$

Then

begin stmt ; stmt end

can be derived as follows:

$$S \Rightarrow \mathbf{begin} \ SL \ \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ SL ; S \ \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ SL ; \mathbf{stmt} \ \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ S ; \mathbf{stmt} \ \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ \mathbf{stmt} ; \mathbf{stmt} \ \mathbf{end}$$

But confusion with English usage leads to a common programming mistake:

begin stmt ; stmt ; stmt ; end

To “legalize” the trailing semicolon, a production was added:

$$S ::= \langle \mathit{empty} \rangle \mid \mathbf{stmt} \mid \mathbf{begin} \ SL \ \mathbf{end} \qquad G_1$$

$$\mid \mathbf{while} \ \mathit{Expr} \ \mathbf{do} \ S$$

Now the “error” has a derivation

$$S \Rightarrow \mathbf{begin} \ SL \ \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ SL ; S \ \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ SL ; \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ SL ; S ; \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ SL ; \mathbf{stmt} ; \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ S ; \mathbf{stmt} ; \mathbf{end}$$

$$\Rightarrow \mathbf{begin} \ \mathbf{stmt} ; \mathbf{stmt} ; \mathbf{end}$$

But like many solutions, this one causes a new problem:

while expr do
stm

has the same interpretation in G_0 and G_1 , whereas

while expr do ;
stm

is rejected by G_0 but accepted (with presumably unintended meaning) in G_1 .

Pascal:

- semicolons are statement *separators*
- *empty statement* is permitted

C:

- semicolons are statement *terminators*
- *empty statement* is permitted

Both

while $x > 3$ **do** ; (* Pascal *)
 $x := x - 1$

and

while($x > 3$);
 $x = x - 1$; /* C */

are accepted, and both define *infinite* loops.

Modula-2 solves the problem:

- semicolons are statement *separators*
- the empty statement is permitted
- the compound-statement grammar is changed:

$$S ::= \langle \mathit{empty} \rangle \mid \mathbf{stmt} \mid \mathbf{begin} \ SL \ \mathbf{end} \qquad G_2$$

$$\mid \mathbf{while} \ \mathit{Expr} \ \mathbf{do} \ SL \ \mathbf{end}$$

Now the strings

while expr do **while expr do ;**
stmt *stmt ;*
end **end**

have identical meanings.

Note that this solution —providing each compound-statement construct with its own **end**— is the same one that solved the “dangling-**else**” problem (Slide 23).

Types

In HLL’s, every valid expression has a *unique type*.

An expression’s type determines which operations are applicable to it.

Motivations:

- abstraction
- type checking

Abstraction provides values and operations that are closer to the problem domain than the machine’s storage cells and its built-in operations.

HLL’s provide such types as

- arrays
- strings
- records
- lists
- tuples

Most HLL’s also provide type-definition facilities such as

- enumeration
- construction
- abstraction