Algorithmics
Basics - EISTI - ING 1

Ecole Internationale des Sciences du Traitement de l'Information

Variables

Why

- to temporarily store data
- data might be coming from hard disk, or entered by the user, or computed by another part of the program...
- think of a box (the *variable*) which contains a piece of data (its *value*) and that can be accessed through a label (its *name*)

How: declaring a variable

- Syntax:

```
variables
  variable_name: type
```

- Pre-defined types:
  `integer,real,character,string,boolean`

Expressions

Definition

An expression is a set of values connected by operators which is equivalent to a single value.

Operators

Different operators are connected to the types of the values they manipulate:

- integer: $+, -, *, /, div, mod$
- real: $+, -, *, /$
- boolean: NOT, AND, OR
- string: &
- Other:
  - Round brackets (control priority): $(,)$
  - Comparisons (boolean results): $=, \neq, >, <, \geq, \leq$

Besides brackets, there are some implicit priority rules which are specific to each operator.

Variable assignment

Why

Assignment associates a value to a variable (name)

How: assignment statement

Syntax:

```
variable_name ← expression
```

Remarks

- Only the left-hand side of an assignment gets modified
- The expression which is assigned to the variable must be of a type which is compatible with the variable's
- The last assignment of a variable erases its previous value

Boolean variables

Why

- to store a piece of data which can only have 2 values (usually opposed): true or false
- to control a conditional alternative in the instruction flow (implement control structures)

Operators

and	*T*	*F*
T	*T*	*F*
F	*F*	*F*

or	*T*	*F*
T	*T*	*T*
F	*T*	*F*

not	*T*	*F*
	F	*T*

Conditional instructions

Why

Within the execution flow, they make the execution of a given set of instructions depend on the value of a boolean test (or condition).

How: Syntax

```
if Condition then
  Instructions
endif
if Condition then
  Instructions 1
else
  Instructions 2
endif
```

Nested conditional instructions

Why

When the conditional test consists of several interdependent conditions, it is possible to *nest* conditional instructions.

How: syntax

```
if Condition_1 then
  Instructions_1
elseif Condition_2 then
  Instructions_2
...
elseif Condition_n then
  Instructions_n
else
  Instructions_(n+1)
endif
```

Assertions

Why

to improve the readability of an algorithm by adding logical (boolean) expressions which must be always true and which formally describe the instructions.

How: Syntax

```
{boolean expression}
```

Example

```
variables
  temp: integer
  write("Enter water temperature")
  read(temp)
  if temp ≤ 0 then                    {temp ≤0}
    write("Frozen")
  elseif temp < 100 then              {0 < temp < 100}
    write("Liquid")
  else                                {temp ≥ 100}
    write("Steam")
  endif
```

Procedure

Definition:

Set of instructions that perform a given task, which is associated to a name that is used to invoke it at any time.

Syntax:

```
procedure name(formal parameter list)
  local variables
  instructions
endprocedure
```

Procedure

How

- Main procedure: program body

```
program name
begin
    Instructions
end
```

- Subprocedure: set of instructions outside the main procedure
- Subprocedures are explicitly called by using their names and providing real values to their formal parameters (between brackets) which therefore become their effective parameters

```
name(effective parameters list)
```

Functions

Definition

A function is a named set of instructions (just like procedures) which returns a value to the calling procedure or function

Syntax

```
function name(parameter list): return type
  local variables
  instructions
  return ...
endfunction
```

Predefined functions

Examples according to the type

- Numerical:

```
int(n:real)      //integer part
rand(n:integer) //random number between 0 and
    n
sqrt(n:integer)//square root of n
sin(x:real) cos(x:real)
```

- Text:

```
//character number:
  length(s:string)
//substring between n1 and n2:
  substr(s:string, n1:integer, n2:integer)
//ascii value of character c:
  ascii(c:chaine)
//character associated to ascii code n:
  char(n:integer)
```

- ...

Parameter passing

Parameter types

- Input parameters (read): I
- Output parameters (write): O
- Input-output parameters (read-write): IO

Example

```
procedure integerDivision(I a:integer, I b:integer,
    O quotient:integer, O remainder: integer)
procedure inversion(IO s:string)
```

Variable scope

Local variables

- declared within the body of a procedure or a function
- visible only within the body itself

Global variables

- alternative way to pass data among procedures (BAD!!!)
- visible to the whole program

Input/output procedures

Read - input

Enter "external" values (example: keyboard).

Write - output

Communicate "internal" values (example: screen).

Syntax:

```
//write a string of characters:
write(I s:string)

//write a string of characters and go to a new
    line:
writenl(I s:string)

//read a variable (polymorphic procedure):
read(O val:?)
```

Here *val* is a variable of any type (character, string, real,...) which contains an external value generated by the user.

Iteration instruction

How?

"while..do" loop syntax:

```
while Condition do
   instructions
endwhile
```

Example: typing an answer

```
variables answer:string
write("Type an answer")
read(answer)
while (answer ≠ "Yes" and answer ≠ "No") do
   write("Type an answer")
   lire(answer)
endwhile
following instructions
```

Count-controlled loops

Why?

In several cases the main purpose of a loop is to count, that is to repeatedly execute a set of instructions for a definite number of times. In this case, the most suitable control structure is the "for loop".

Syntax

```
for counter ← initial to final step stepValue
  Instructions
endfor
```

- initial contains the initial value of the counter
- final contains the final value of the counter
- stepValue contains the value of the increment of the counter after every cycle

Loop invariant

Definition

A specific assertion associated to a loop: it is a boolean formula which is true at any iteration of the cycle.

Why

Loop invariants are of paramount importance to formally prove the final result of a loop.

Loop invariant

Example: Euclidean division

```
B ← b
R ← a
Q ← 0
{a = B * Q + R}
while R ≥ B do {(a = B * Q + R) ∧ (R ≥ B)}
  R ← R − B
  Q ← Q + 1
endwhile
{(a = B * Q + R) ∧ (R < B)}
```

Invariant proof

- Initial condition: $a = b * 0 + a = a$
- Let R', B', Q' be the values modified by the loop on R, B, Q:
  - $R' = R - B$ and $Q' = Q + 1$
  - therefore $B' * Q' + R' = B * (Q + 1) + R - B = B * Q + B + R - B = B * Q + R$
  - besides, $R - B$ strictly diminishes (if $B \neq 0$)
- At the end of the loop, $a = B * Q + R$ and $R < B$ $\square$

Loop invariant

Example: power computing

```
A ← a
N ← n
R ← 1
{A^N * R = a^n}
while N > 0 do
   if even(N) then
      A ← A*A
      N ← N/2
      {(A^N * R = a^n)}
   else
      R ← R*A
      N ← N–1
      {(A^N * R = a^n)}
   endif
endwhile
{(A^N * R = a^n) ∧ (N = 0)}  so  {R = a^n}
```

Algorithm cost

Definition

The cost of an algorithm can be defined as the number of instructions which are executed to get the desired result.

- very simple calculation for sequential instructions (just count them for the worst case!)
- can be more tricky for loops (worst-case scenarios can be hard to determine):
  - "for loop": the counter
  - "while..do": how many iterations before the condition becomes false?
  - nested loops: depends on the nesting level and the mutual interdependence of stop conditions for the loops

Algorithm cost

Example: Euclidean division

```
B ← b    // 1 instruction
R ← a    // 1 instruction
Q ← 0    // 1 instruction
{a = B * Q + R}
while R ≥ B do          // 1 instruction
   {(a = B * Q + R) ∧ (R >= B)}
   R ← R − B                    // 2 instructions
   Q ← Q + 1                    // 2 instructions
endwhile                        // Q loops
{(a = B * Q + R) ∧ (R < B)}  // Total: 5 * Q + 3
```

The cost must only depend on the input parameters a and b. Therefore the cost Q of the algorithm is function of a and b:

- according to the final assertion, $a = B * Q + R$ and $R < B$
- therefore $Q = (a - R)/B < (a - B)/B = (a - b)/b$
- Cost's upper bound is $5 * (a - b)/b + 3$