5 - abstraction and parameter passing

abstraction = make a program reusable by enclosing it in a body, hiding the details, and defining a mechanism to access it.
↳ separating the usage and implementation of program segments
ex: programming languages over machine language
☆ declare once, use many times
☆ code reusability

abstraction types

function and procedure abstraction =
- functions are abstractions over expressions
- procedures are abstractions over commands (exists because of their side effects)
  ↳ no value but contains executable statements as detail (void functions of C)

selector abstraction = [ ] operator selects elements of an array

generic abstraction = abstraction over declaration
- same declarion pattern applied to different data types (template functions in c++)
  template <class T> : type of T declared at compile time
  ↳ for every different T (int, float...) it declares different functions
  ↳ it is different from polymorphism (done at run time) (slower)
  So it is more efficient since everything done at compile time

iterator abstraction = iteration over a user defined data structure
↳ in C++, begin() and end() functions of linked list.

entity	abstraction
expression	function
command	procedure
selector	selector function
declaration	generic
command block	iterator

parameters =
formal parameters = a variable and its type as they appear in the prototype of the function
actual parameters = the variable or expression corresponding to a formal parameter that appears in the function in calling environment
modes: in = passes info from caller to calle
out = calle writes values in caller
in/out = caller tells callee value of variable, which may be updated by callee

parameter passing mechanisms

Copy mechanism: ...

copy mechanism: assignment based

↳ C only allows copy-in mechanism. → is called as pass by value.

binding mechanism: based on binding of the formal parameter variable/identifier to actual parameter value/identifier

↳ constant binding: formal parameter is constant during the function (in Haskell)

↳ variable binding: formal parameter variable is bound to the actual parameter variable. same memory area is shared by two variable references → also known as pass by reference

pass by name: substitution based

textually substitute the argument in a procedure call for the corresponding parameter in the body of the procedure

↳ the argument expression is re-evaluated each time the formal parameter is passed (normal evaluation)

↳ the procedure can change the values of variables used in the argument expression and hence change the expression's value (in imperative languages not in Haskell)

evaluation order

normal order evaluation: actual parameter is re-evaluated each time it is used

↳ has an advantage ex: x y = if x>0 then y else x
if this is not true y is not evaluated (y can be like 0/5)

↳ some parameters evaluated once or more, or not evaluated at all

↳ side effects are repeated

eager evaluation: actual parameters are evaluated first, then passed

↳ faster than normal evaluation

↳ evaluates parameters exactly once

church-rosser property = if a function does not have side effect, normal order and eager (or others) evaluation produce same result (order of the evaluation does not matter)

↳ then more efficient solution: lazy evaluation

lazy evaluation: faster, computes the same result

↳ evaluates the parameter first time that is needed (does not reevalu

↳ so, evaluates parameters at most once

ex: Haskell, infinite values

corresponding principle = if for each form of declaration there exists a corresponding parameter mechanism, the PL satisfies this principle

C/C++ ✓ pascal ✗