cc@cse.concordia.ca EV3.229. Monday
users.encs.concordia.ca/~cc/348/

Prolog Queries

statement : parent(peter, daphne) → ask?
query : ?- parent(peter, daphne)

variable in queries — always start with a capital letter
?- parent(X, daphne) (who is parent of daphne?)
X = james ; → continue search
X = peter ;
No

fact that can be deduced not necessarily in the database, can use rule to define it
$G = \forall x \forall y \forall z ((p(X,Z) \wedge p(Z,Y)) \rightarrow g(X,Y))$
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
(, = logic And, conjunction)

ex: ?- grandparent(roger, daphne)
⇓ unification + instantiation
parent(roger, Z) AND parent(Z, daphne)
ex: ?- grandparent(X, daphne).
X = judy ;
X = mark ;
No.

isAncestorOf relation : two rules
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

☆ rules can be build not only based on other predicates, but also other rules.

anonymous variable : underscore character _

Prolog programs consist of collections of statements
All statements are constructed from terms
1. Constants (numbers, atoms)
2. Variables
3. Compound terms : functor [argument +]

Statements are grouped into procedures
↓
define relationship between arguments
consist of one or more assertions,
or clauses
facts rules

- Facts : propositions that assumed to be true

Ground queries : is it the case that a given statement is true?
Non-ground queries : under what conditions, if any, is a given statement true?

unification (or matching)
two terms unify if subsitiution can be made for any variables in the terms so that the two terms can be made equal
constants unify with other constants
variables instantiate to values

- Rules
head :- body
conclusion :- condition
consequent :- antecedent - expression

→ one procedure (18 facts)

eg: parent (tom, adam). parent (tom, helen). ... parent (peter, daphne)
grandparent (X, Z) :- parent (X, Y), parent (Y, Z).
→ one procedure (1 rule)

search ?- grandparent (judy, daphne).
a) unify
b) instantiating X to judy, Z to daphne
c) resolve the two queries parent (judy, Y), parent (Y, daphne)

Anonymous variables —
is_father(X) :- father (X, _).
is_mother(X) :- mother (X, _).

List A list is a finite ordered sequence of zero or more elements that can be repeated.
can only access : first element → head
the list made up of all except the head → tail

a ≠ <a> ≠ <<a>>
element list list, element is a list
↓ ↓
head is a head is <a>,
tail is tail is <>

ex 1. define a clause "first" which succeeds if an element is the head of list
first (F, [F|_]).
ex 1.1 under what conditions an element is the head of list [a b c]
?- first (F, [a, b, c]).
F = a

Hilroy

ex3 how to define a clause that check element is in the list?

```
member (X, [X|_]).
member (X, [_|T]) :- member (X, T).
```

ex5 define a rule "last" if an element is the last element of a given non-empty list.

case 1. List has only one element

```
last (L, [L]).
```

case 2. List has more than one element

```
last (L, [H|T]) :- last (L, T).
```

ex4. define clause "add" if a new list can be created when placing the element as the head of another list.

```
add (X, L, [X|L]).
    ?- add (a, [b, c, d], NewList).
    NewList = [a, b, c, d].
    ?- add (a, [b, c], [a, b, c]).
    Yes
```

ex7. 2a → b

```
a2b([], []).
a2b([a, a|Ta], [b|Tb]) :- a2b(Ta, Tb).
```

function : return sth
procedure : without return

function :
```
int f(..)
{
    ...
    return ...
}
```

procedure :
```
void f(..)
{
    ―
}
```

Boolean operation :

- Conjunction : X ∧ •
- Disjunction : + ∨
- Inverse : ' ~ ¬

Define procedures to represent logical operators :

operation (in, out)
  input → in, output → out

ex. inverse (0, 1) → The inverse of 0 is 1
  or (0, 1, 1) → The disjunction of 0 and 1 is 1
  and, or, inv, nand (not and), nor (not or), xor (exclusive or)

ex. circuit (X, Y, Out) :- inv (Y, Temp1),
  and (X, Tmp1, Tmp2),
  or (Tmp2, Y, Out).

$(X \wedge \neg Y) \vee Y$ (X AND Y') OR Y

?- circuit (X, Y, 0).
X = 0,
Y = 0;
false

Factorial

factorial (0, 1).   X > 0,
factorial (X, R) :- factorial (X', R'), X' is X-1, R is X*R'.

Lab 2

- Define list-length(L, R) to mean R is the length of list L

```
list-length([], 0).
list-length([_|T], R) :- list-length(T, R1), R is R1+1.
?- list-length([1, 2], R).
① x
② list-length([_|[2]], R) :- list-length([2], R1), R is R1+1.
① x
② list-length([_|[]], R1) :- list-length([], R1'), R1 is R1'+1
① list-length([], R1').
                R1' = 0. → R1 = 1. → R = 2.
```

- Define append1(L1, L2, L3) to mean L1 and L2 forms a new list L3

```
append1([], L, L).
append1([H|T], L, [H|R]) :- append1(T, L, R).
? append1([1, 2], [3], L3).
① x                                      → L3 = [1|R]
② append1([1|[2]], [3], [1|R]) :- append1([2], [3], R).
① x                                      → R = [2|R']
② append1([2|[]], [3], [2|R']) :- append1([], [3], R').
① append1([], [3], R').
        ↳ R' = [3] → R = [2, 3] → L3 = [1, 2, 3]
```

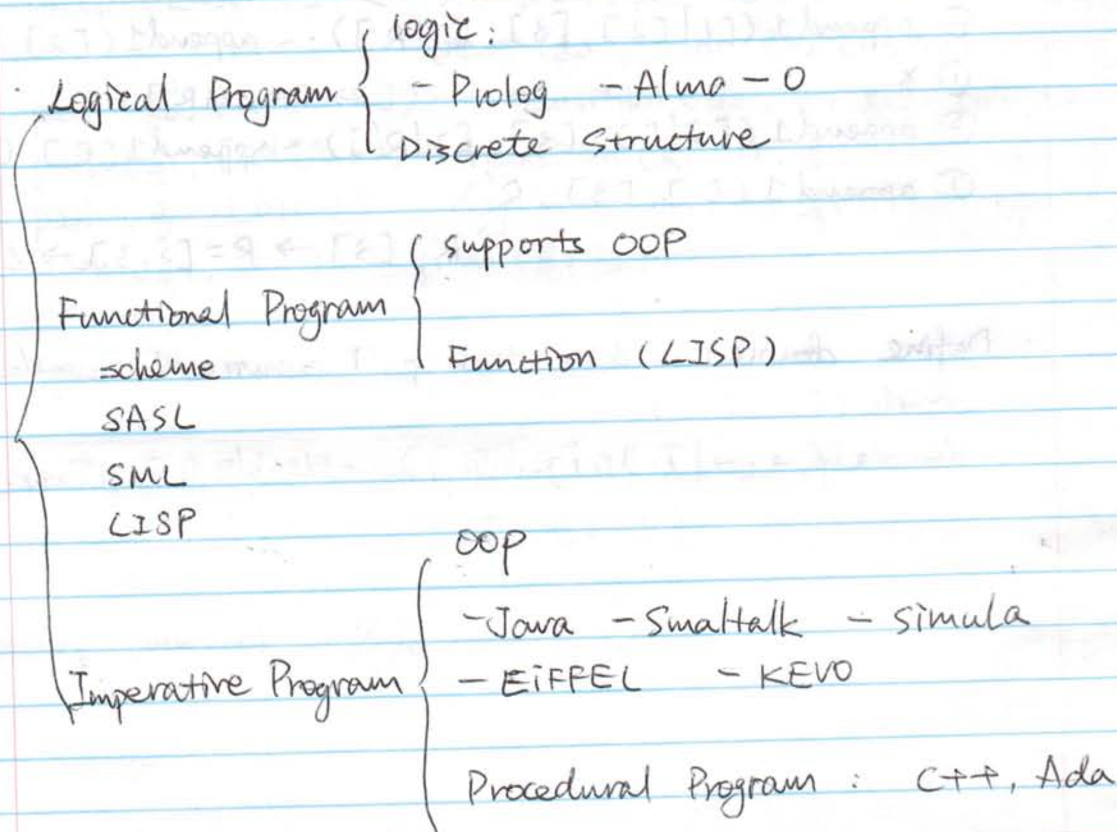
- Define double(L1, L2). true if L1 is twice the length as L2

```
double([], []).
double([_, _|T1], [_|T2]) :- double(T1, T2).
```

Functional programming

P17 Programming Paradigms Fundamental Style of Programming

- Abstraction
  - is a world with a quite general meaning and helps us to manage the complexity
  - is useful for describing a P. L. in general terms without details of every construct
    - classes
    - procedures
    - functions
    - data types
- Computation (≡ semantic model)
  describes effects of various operations without describing the actual implementation



LISP: programming language to model, construct and manipulate lists

```
( )               empty list
(1 3 5 7)         list of four element, numbers 1 3 5 7
((1 2)(3 4))      list of two elements, list (1, 2) & list (3 4)
(((1 2)(3 4)))    list of one element, list ((1 2)(3 4))
(a (b 1) 2)       list of three elements, symbol a, list & nb 2
```

Data type

- simple : boolean, number, symbol
- composite : list

Prefix: a form of notation for logic, arithmetic, and algebra. It places operators to the left of their operands

ex: 14 − (2×3) ⟶ (− 14 (× 2 3))

$\frac{a - b \times c}{d \times e + f}$ ⟶ (/ (− a (* b c)) (+ (* d e) f))

Arity: used to describe the number of arguments or operands that a function takes.

unary function (arity 1) takes one argument
binary function (arity 2) takes two arguments
ternary function (arity 3) takes three arguments
variable arity functions can take any number of arguments

quote
' (. . .) → return (. . .) not evaluated

```
> (let ((x 5))
(or (< x 2) (> x 3)))
T    (T means true, nil means false)
> (let ((x 5))
(and (< x 7) (< x 3)))
NIL
> (or (and (= 1 1) (< 5 6)) (not (> 3 1)))
T
```

Three (built-in) functions to create list

- cons : create a list by adding element ~~as~~ the head of existing list
- list : create a list comprised of its arguments
- append : create a list by concatenating existing lists

- cons (h, L) → head is h, tail is L

ex: cons (a, <>) = <a>

cons (a, <b, c>) = <a, b, c>

```
(cons 'a '())                          return  (a)
(cons 1 '(2 3))                                (1 2 3)
(cons '(1 2) '(3 4))                           ((1 2) 3 4)
(cons 'a (cons 'b '()))                        (a b)
(cons 'a (cons (cons 'b (cons 'c '()))
               (cons 'd (cons 'e '()))))
                                        return  (a (b c) d e)
```

if no ', ← lisp will try to evaluate and give error

```
(cons 'a '()) ≡ (cons 'a nil) → (A)
(cons 'a (cons 'b '())) ≡ (cons 'a (cons 'b nil)) → (A B)
(cons 'a (cons (cons 'b (cons 'c '())) (cons 'd (cons 'e '()))))
    → (a (b c) d e)
```

- list : takes any number of arguments and return a list composed of these arguments

```
(list 1 2 'a 3)            return  (1 2 A 3)
(list 1 '(2 3) 4)                  (1 (2 3) 4)
(list '(+ 2 1) (+ 2 1))            ((+ 2 1) 3)
(list 1 2 3 (list 'a 'b 4) 5)      (1 2 3 (a b 4) 5)
```

- append : takes any number of list arguments

```
(append '(1 2) '(3 4))                    (1 2 3 4)
(append '(1 2 3) '() '(a) '(5 6))         (1 2 3 a 5 6)
(append '(1 2 3 '(a b c)) '() '(d) '(4 5))  (1 2 3 (QUOTE
                                              (a b c)) d 4 5)

(append 1 '(4 5 6))
Error : 1 is not of type LIST.
```

(append (list 1) '(4 5 6)) → (1 4 5 6)

operations:

car / first   cdr/rest

(car '(a s d f)) return a

(car '((a s) d f)) (a s)

(cdr '(a s d f)) (s d f)

(cdr '((as) d f)) (d f)

(cdr '((as) (df))) ((d f))

access the second element ?

(car (cdr '(1 (3 5) (7 11)))) return (3 5)

- (car (list '() '(a b c))) return NIL since ( ) empty list
- (cdr (append '() '() '())) return NIL
- (cons '(+ 2 3) (list 'f (cons 'g '()))) → ((+ 2 3) f (g))
  length = 3
- (append (list 'b '(d e) (* 2 3)) (cons '(+ 2 3) (list 'f (cons 'g '()))))
  → (b (d e) 6 (+ 2 3) f (g)) length = 6
- (list (append '(+ 1 4) '() (list '() '())) (cons (+ 1 4)
  (list 'a (cons (+ 1 7) '()))))
  → (((+ 1 4) () ()) (5 a (8))) length = 2
- (car (cdr (cdr (append (append '() '(a) '()) (list 'b '()
  (cons (+ 3 4) '())))))) → NIL
  → (a b NIL (7))

function listp return true if its argument is a list

(listp '(a b c)) return T

(listp 7) NIL

Hilroy

- numberp
- zerop
- evenp
- oddp

- mathematical operation :

(sqrt a) return $\sqrt{a}$

(expt a b) a^b

(log a) log a natural

Control flow

- single selection : (if test
  then
  else)

ex. (if (listp '(a b c))
(+ 3 7)
(+ 1 3)) return 10

- multiple selection : (cond (question answer)
  ...
  (else answer) ; optional
)

- For the first condition that evaluates to true, LIST evaluates the corresponding answer, and the value of the answer is the value of the entire cond expression
- If the last condition is else and all other conditions fail, the answer for the cond expression is the value of the last answer expression
- can also use t (true) in place of else

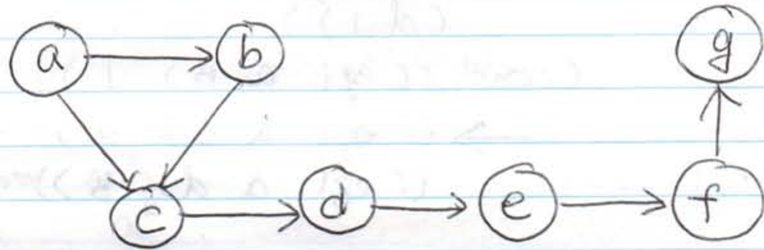
ex. (let ((a 1)
(b 2)
(c 1)
(d 1))
(cond ((eql a b) 1)
→ ((eql a c) 2) return 2
((eql a d) 3)))

Lab 3

2. one rule: l([X|Y]). The query l([]) will return false.
3. l([_|_]). The query l([a]) will return true.
4. In Prolog, the answer "no" or "false" always means that the goal was proved false. X
5. a(X) :- b(X), !, c(X).
b(1)
b(2) ↓ take the last instiantiation for the variable
b(3) and fix it, no none changes allowed.
c(2)
what is the answer of the query ?- a(X)? false.
if no "!", return: X=2;
false.
6. write prolog return true if the list it takes has an even size
Let assume empty list has length even.
size_even([]).
size_even([_,_|T]) :- size_even(T).

7. a non-cyclic graph, write prolog pathk (X, Y, K) return the paths of a specific size k. eg. if execute ?- pathk(X,Y,3), prolog will return

X=a
Y=d;
X=b;
Y=e;
X=a,
Y=e;
X=c,
Y=f;
X=d,
Y=g;
false



define database as edge(a,b) ....

pathk(N1, N2, 1):- edge(N1, N2).
pathk(N1, N2, K):- K>1, edge(N1, N),
K1 is K-1, pathk(N, N2, K1).

8. range(I,K,L). I<=K. construct list L that consists of integers between [I, K] inclusively.
ex. range(1,5,L) ⇒ L=[1, 2, 3, 4, 5].

① range(I, I, [I]).
② range(I, K, [I|L]):- I<K, I1 is I+1,
range(I1, K, L).

?- range(1, 3, R).
① X [1,2,3] =2 [2,3]
② range(1, 3, [1|L]):- 1<3, I1 is 1+1, range(2, 3, L).
① X [2,3] =3 [3]
② range(2, 3, [2|L']):- 2<3, I1' is 2+1, range(3, 3, L').
① range(3, 3, L'). L'=[3]

9. size(L, N). N = size of L.
① size([], 0).
② size([_|T], N) :- size(T, N1), N is 1+N1.

ch9. Define function: (defun name (parameterlist) body)

ex. (defun absdiff (x y)
  (if (> x y)
    (- x y)
    (- y x)))

ex. return third element of list
  (defun third2 (lst)
    (car (cdr (cdr lst))))
  (cddr)

side effect: a function or expression modifies some state in addition to its return value.

Pure fuction: ① always return same result given same argument
② doesn't cause any semantically observable side effect

ex. length(string) → pure
today() → impure
print(arg) → impure, cause output as effect

Pure functions allow optimization:
$y = f(x) \times f(x)$ if $f(x)$ is pure, then $z = f(x)$, $y = z \times z$

- referentially transparent if the expression can be replaced with its value without changing the program, i.e. yielding a program that has the same effects and output on the same input
- All referentially transparent functions are determinate.
- If all functions involved in the expression are pure functions, then the expression is referrentially transparent. In pure functional programming, referential transparency is enforced for all functions.

Variable binding : (let
((binding$_1$)
(binding$_2$)
...)
(expression))

nested binding : outer become invisible

```
(let ((x 1))                 x is 1
    (let ((x (+ x 1)))       x is 2
    (+ x x)))                return 4
```

let* variable depend on other variable

```
(let* ((x 10)
      (y (* 2 x)))           return 200
      (* x y))
```

referential transparency : (* 5 5) replace by 25
sin(x) alway give same result for any given x

not hold : x++
system.out.println("Hello")
today()

Being side-effect free is necessary but not sufficient for referential transparency. needs to be determinate too

Idempotence : same effect if used multiple times as it does if used only once
ex. abs(x) = abs(abs(x)) = abs(abs(abs(x))) = ...

Higher-Order function : at least one following :
① take one or more functions as their arguments
② return a function

ex. derivative function

ex. (sort (list 5 0 7 3 9 1 4 13 23) #'>)
return (23 13 9 7 5 4 3 1 0)

ex. (mapcar #'* '(2 3) '(10 10))
return (20 30)

ex. (funcall #'+ 1 3 4)
return 8

ex. (apply #'+ 3 4 '(1 3 4)) last argument is list
return 15

Function composition $f \circ g(x) = f(g(x))$

ex. $f(x) = x+2$, $g(x) = x^2-1$

$(f \circ g)(x) = (x^2-1)+2$

ch11. Recursion

① one or more base cases

② one or more recursive cases

sum :
```
(defun sum (lst)
  (if (null lst)
      0
      (+(car lst) (sum (cdr lst)))))
```

```
(defun sum (lst)
  (cond ((null lst) 0)
        (t (+ (car lst) (sum (cdr lst))))))
```
↑ t: always true!

last element of list:
```
(defun last2(lst)
  (if (null (cdr lst))
      (car lst)
      (last2 (cdr lst))))
```

Hilroy

reverse list : (defun reverse2(lst)
(if (null lst)
(nil)
(append (reverse2(cdr lst)) ^list(car lst))))

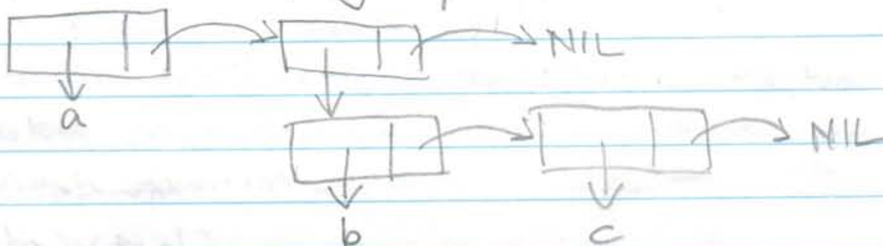
cub all elements in list :
(defun cub (lst)
(if (null lst)
'( )
(cons (* (* (car lst) (car lst)) (car lst))
(cub (cdr lst))))))

Lab

2. Evaluate the expression
not ((1+2=3) ∧ (4<3) ∧ (2+3=7))
(not (and ((= (+ 2 3) 7)
(< 4 3)
(= (+ 1 2) 3))))

3. set a variables PETS with 'DOG and 'CAT
(SETQ PETS '(DOG CAT))
(not nil) → true
(not (member 'dog pets)) → nil
(not (member 'tiger pets)) → t
(and t t nil) → nil

4. construct a list using keyword "cons" :



(a (b c))

(cons 'a '((b c))) → (a (b c))
(list 'a '(bc)) → (a (bc))
(append '(a) '(b)) → (a b) drop one parenthis

so (cons ('a cons ('b cons 'c '()))) NoX
(cons ('a (cons (cons 'b (cons 'c 'c)) '()))))

6. (cons 'a '(bc)) → (a b c)
(cons '(a b) '(c d)) → ((a b) c d)
(cons 'a ()) → (a)
(cons () '(a b c)) → (NIL a b c)
(cons 'a (cons 'b (cons 'c '(d))))
(list '(+ (1 + 4) (* 4 4))) → ((+ (1 + 4) (* 4 4)))
(list '(a b) '(ab)) → ((a b) (a b))
(list '(a b) (cons 'a '(b)) (cons '(a b) ()))

(list (/ 14 (float 4)) () '(a))
(append '(a b) '(d e)) → (a b d e)
(append '() '(a b d e)) → (a b d e)

7. (cons (car '(a b c)) (cdr '(a b c)))
(car (cons 'a '(b c))) → a
(cons (car (append (cdr '(ab)) '(d e)))) → error!
ap ('(b) '(d e)) → (b d e)
car (b d e) → 'b → cons need 2

9. define function reture two times the input value
(defun double (X)
(* X 2)) then (double 3) → 6

Hilroy

format t "name ~A, number ~D, section ~A",
"comp", 348, "u")

ex. remove first occurrence
ex. remove all occurrence
ex. merge two lists

☆ guidelines on defining functions
① unless function is trivial, break the logic into cases with cond
② when handling list, normall adopt a recursive solution
Treat the empty list as a base case
③ operate on the head of a list and recur on the tail of the list
④ to delete the head, simply recur on the tail
⑤ to keep head, use cons to place it as the head
⑥ use else or t to cover remaining cases

Ch12 setunion

```
(defun setunion (lst1 lst2)
    (cond ((null lst1) lst2)
          ((null lst2) lst1)
          ((member (car lst1) lst2) (setunion (cdr lst1) lst2))
          (t (cons (car lst1) (setunion (cdr lst1) lst2)))))
```

bag-to-set

```
(defun bag-to-set (bag)
    (cond ((null bag) '())
          ((member (car bag) (cdr bag)) (bag-to-set (cdr bag)))
          (t (cons (car bag) (bag-to-set (cdr bag))))))
```

Lab 5.

1. Give the output for Lisp :

① cdr '(10 (20) (30))

((20)(30))

② cdr (cdr '(10 (20) (30)))

((30))

③ car (cdr (cdr '(40 (50 (20) (30)) (60 (80) (40)))))

(60 (80)(40))

2. ~~output~~ give the value

Value is what return
output is what print

a)
```
(loop
    (print "Lisp loop")
    (return 5)
    (print "Lisp block"))
```
5

b)
```
(let ((n 1))
    (loop
      (if (> n 4) (return))   --> return nothing so NIL
      (print n)
      (* n 2)   -> no assign
   (incf n)))
```
NIL

c)
```
(let ((n 1))
    (loop
      (if (> n 4) (return (list n)))
      (print n)
      (* n 2)
      (incf n)))
```
(5)

Hilroy

3. write a recursive Lisp function last1 that returns the last element of a list

```
(defun last1 (lst)           → null ok too
    (cond ((equal nil lst) nil)
          ((equal (cdr lst) nil) (car lst))
          (t (last1 (cdr lst)))))
```

Note: eql: only test structures
= only test numbers, ex. (= length(lst) 1)
equal both

4. write notlast that takes as argument a list and returns the same list but without the last element

```
(defun notlast (lst)
  (cond ((equal nil lst) nil)
        ((equal (cdr lst) nil) nil)
        (t (cons (car lst) (notlast (cdr lst))))))
```

ex. (notlast '(1 2))

① X
② X
③ (cons '1 (notlast '(2))) ⇒so (cons '1 nil) → (1)

① X
② nil

5. write reverse2 that takes as argument a list and returns its reverse

```
(defun reverse2 (lst)
    (cond ((null' lst) '())
          (t (append (reverse2 (cdr lst)) (list (car lst))))))
```

6. write reverse3 that reverses a list but also internal lists

```
(defun reverse3 (lst)                    (listp (car lst)) check list
  (cond ((null lst) '())
        ((listp (car lst)) (append (reverse3 (cdr lst))
                                   (list (reverse3 (car lst)))))
        (t (append (reverse3 (cdr lst)) (list (car lst))))))
```

ex. (reverse3 '((1 2) 3))

① X

② append (reverse3 '(3) (list (reverse3 '(1 2)))) → (3 '(2 1))

↳ ① X   ↳ ((2 1))

② X

③ (append (reverse3 '()) (list 3)) → (3)

① '()   (3)

7. define consecutive that returns true only if the list it takes as argument has consecutive elements only, which means it has form [x, x+1, x+2, x+3, ..., x+n] n>0
can assume element are numbers

```
(defun consecutive (lst)
  (cond ((and (null (cdr (cdr lst))) (equal (car lst) (- (car
                                      (cdr lst)) 1))) t)
        ((equal (car lst) (- (car (cdr lst)) 1)) (consecutive
                                                   (cdr lst)))
        (t nil)))
```

Lab

```
(defclass videogames (software)
  ((name : accessor vgname
         initarg : name
         initform 'portal)
   (subject : reader vg-sub
            initarg: subject
            initform 'confusion)))

(defmethod powzor ((vg videogames))
  (setf (vg-name vg) 'gg))

(defmethod powzor : around ((vg videogames))
  (setf (vg-name vg) 'notgg))

(setf vg (make-instance 'videogames))
```

① considering the affects of standard method

```
(defclass food() ())

(defclass pie (food)
  ((filling : accessor pie-filling
            : initarg : filling
            : initform 'apple)))

(defmethod cook ((p pie))
  (print "cooking a pie.")
  (setf (pie-filling p) (list 'cooked (pie-filling p))))

(defmethod cook : before ((p pie))
  (print "A pie is about to be cooked."))
```

```
(defmethod cook :after ((p pie))
    (print "A pie has been cooked."))

(setf pie-1 (make-instance 'pie :filling 'apple))
```

what is the output : (cook pie-1)?

"A pie is about to be cooked"
"Cooking a pie"
"A pie has been cooked"
(cooked apple) ⟶ return

```
(defmethod cook :around ((f food))
   (print "Begin around food.")
   (let ((result (call-next-method)))
     (print "End around food.")
     result))
```

⟶ if parameter is type "pie", then this has no precedence, it will execute after "before", but here because superclass has precedence, so it execute before "before"

then :

"Begin around food"
"A pie is about to be cooked"
"cooking a pie"
"A pie has been cooked"
"End around food"
((ooked (cooked apple))

if parent & child have "before, around, after", then

Hilroy

Lab Ruby www.compileonline.com/execute_ruby_online.php

Ex1.
```
def welcome (name)
  puts "Hi #{name}"
end
```

Ex2.
```
def multiply (a, b)
    c = a * b
    return c
end
```

Ex3.

1	2	3
5	2	7
4	6	10
3	4	6

Ex4.
```
def getCostAndMpg
    cost = 30000
    mpg = 30
    return cost, mpg
end
```

Ex5.
```
presidents.each {|i| print i "\n"}
```

Ex7.
```
class Box
    def initialize (w, h)
        @width, @height = w, h
    end
    def printWidth                def setWidth (w)
        @width                        @width = w
    end                           end
end
```