Prolog Programming Language

Relations/Predicates

predicates are building-block in predicate calculus $p(a_1,a_2,\ldots,a_k)$
relation: we attach meaning to predicate but within logical syst they are simply structural building blocks with no meaning beyond that provided by explicitly-stated interrelationships
ex parent(X, Y): X is a parent of Y
ex mother(X, Y): X is mother of Y so in prolog:

```
mother(X, Y) :-
        parent(X, Y),
        female(X).
```

- "," means and (conjunction)
- ":-" means if (implication)
- ";" means or (disjunction)

recursion

"I am my own grandpa"
Transitive close
Ex: graph declared with facts

```
        Edge(1,2). Edge(2,3). Edge(2,4). Reach(X,Y) :- edge(X,Y).
```

prolog execution

Call: call a predicate (invocation)
Exit: return an answer to the caller
Fail: return to caller with no answer
Redo: try next path to find answer

Syntax of prolog programs

Program = sequence of clauses
Each clause is of the form head :- body.
Head = one term
Body = list conjunction of terms
Fact = Clause with empty body ex raining(ny)
Rule = Clause can sometimes be called a rule ex wet(X) :- raining(X).

Logic programming concepts

- Operators: conjunction, disjunction, negation, implication
- Universal and existential quantifiers
- Statements: T, F, unknown.
    - Axioms: assumed true.
    - Theorems: probably true.
    - Goals: we'd like to prove true
- All statement are in form of HORN CLAUSES = HEAD + BODY

- Term: can be constant, variable, or structure of a functor+ parenthesize list of arguments
- Meaning of a rule: the conjunction of the terms in the body implies the head.
- Query/top-level Goal = clause with an empty head ex: ?- wet(X).
- Prolog interpreter has collection of facts and rules in its database = fact are axioms (thing interpreter assumes to be true) & prolog provides automatic way to deduce true results from facts and rules.
- A structure can play the role of a data structure or a predicate
- A constant is either an ATOM (looks like an identifier beginning with a lowercase letter, or a single quoted charact string) or a number (looks like an int or real from some more ordinary language)
- Variable looks like an identifier beginning with an upper-case letter
- Rules are theorems that allow interpreter to inger things
- Variables whose first appearance is on the left hand side of the clause have implicit universal quantifiers
- Variables whose first appearance is in body of clause have implicit existential quantifiers

Prolog programs

- Atomic data:
  - Numeric constants : int , float
  - Atoms: strings of charact enclose in single quotes ex 'Concordia'
  - Identifiers: sequence of letters, digits, underscore, beginning with a lower case letter ex mark, r2d2, one_element
- Variables
  - Are denoted by identifiers beginning with an Uppercase letter or underscore ex X, Index, _param
  - These are single-assignment logical variables
    - Be assigned only once
    - Diff occurrences of same variable in a clause denote same data
    - Variables are implicitly declared upon first use
      - They are not typed
    - if doesn't start with "_" then assumed it will appear multiple time in the rule
  - Anonymous variables/Don't care variables: beginning with "_"
    - Underscore by itself = variable
      - Each occurance will correspond to diff variable
      - A variable with "_" and with charact: descriptive name & used to create relationships within a clause (used 1+)
  - Warning used to identify bugs
- Queries
  - To run prolog program, one ask interpreter a question(query) which interpreter tries to prove
  - Declarative meaning: what are the logical consequences of program?

- Procedural meaning: for what values of variables in query can I prove query?
  - Use gives syst a goal = syst attempts to find axioms + inference steps to prove goal

Procedural meaning of prolog

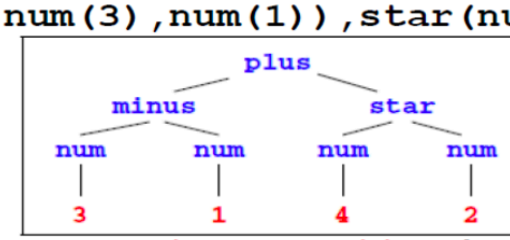
- Prolog interpreter works by what is called backward chaining (top-down, goal directed)
  - Begins with thing it is trying to prove and works backwards looking for things that would imply it until it gets to facts
- Also possible to work forward but can be very time consuming
- Interpreter starts at beginning of database & looks for smthing with which to unify the current goal
  - If finds fact = success
  - If finds rule = attempts to satisfy terms in body of rule depth first
  - Process is motivated by resolution principle of Robinson
- When attempts resolution = interpreter pushes current goal onto stack, makes first term in body current goal, and goes back to beginning of database and starts looking again.
- If fails: interpreter undoes unification of left hand side & keeps looking through database for smthing else which to unify == BACKTRACKING
- Prolog isn't purely declarative

Structures

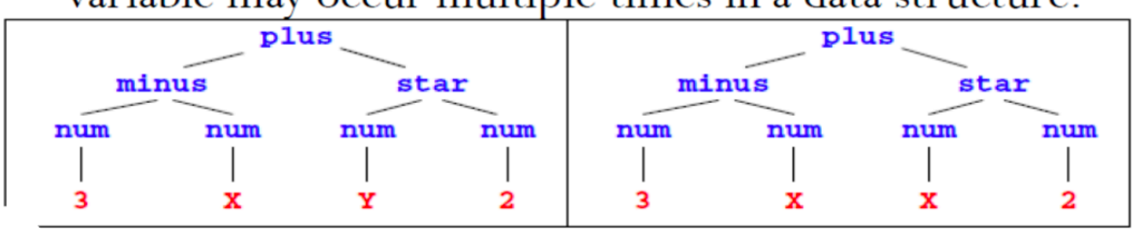
- If f is an identifier and t_1, t_2, …,t_n are terms then f(t_1, t_2, …,t_n) is a term. F is a function and t is an argument
- Structures are used to group related data items and to construct trees

Trees

```
plus(minus(num(3),num(1)),star(num(4),num(2)))
```



- Data structures may have variables. And the same variable may occur multiple times in a data structure.



Matching

- Given 2 terms, we can ask if they match each other. Rules:
  - A constant matches with itself
  - Variable matches with anything
    - Ex A = 35, A=B, then B becomes 35
  - 2 structures match if they have: same functor, same arity, match recursively
- General rule whether S and T match are:
  - If S and T are constants
  - If S is variable and T is anything
  - If T is variable and S is anything
  - If S and T are structures: same functor, corresponding arguments components have to match.
- Matching is predominant means for accessing a structures arguments

Declarative and procedural Way

- Prolog programs can be understood 2 ways: declaratively and procedurally
  - P :- Q,R
  - Declaractive way: P is true if Q and R are true
  - Procedural Way: to solve P first solve Q then R OR to satisfy P, first satisfy Q then R

Lists

- Prolog uses special syntax to represent and manipulate lists
  - [1,2,3,4]: represents a list
  - [1 | [2,3,4]] : head is first element, tail is the rest
  - empty list : [] or nil
- List are special cases of trees
- Strings: a sequence of charact surrounded by quotes is equivalent to a list of numeric charact codes: "abc", "to be, or not to be".

Programming with Lists

- Ex: member/2 to find if a given element occurs in a list:
  - The program: member(X, [X|_]). member(X, [_|Ys]) :- member(X, Ys).
  - Ex of questies: ?- member(2, [1,2,3]).
- Append/3: concatenate two lists to form the third list:
  - Append([], L, L).
- Is the predicate a function? No. not applying arguments to get result. Instead we're proving that a theorem holds. => can leave other variables unbound
- Append example trace
- Len/2 finds the length of a list (first argument)
  - Len([], 0). Len([_Xs], N+1) :- len(Xs, N).
  - Queries: ?- len([]), X). X = 0

Arithmetic

- ?-1+2 = 3.   False
- in predicate logic, basis for prolog, the only symbols that have a meaning are the predicates themselves
- in particular, function symbols are uninterpreted: have no special meaning and can be used to construct data structures.
- Meaning for arithmetic expressions is given by the built-in predicate “is”:
    - ?- X is 1 + 2. Succeeds, binding X = 3.
    - ?- 3 is 1 + 2. Succeeds.
- General form: R is E where E is an expression to be evaluated and R is matched with expression’s value.

Conditional Evaluation

- Conditional operator: the if-then-else construct in Prolog:
    - If A then B else C is written as (A-> B; C)
    - In prolog it means: try A. if can prove it, go prove B and ignore C. If A fails, go on to prove C ignoring B
- Computation of n!   factorial(N, F) :- …
    - N is input parameter; F is output parameter
    - Body of rule species how output is related to input
        - If N =< 0 then F=1   if N >= 0 then F = N*factorial(N-1)

Imperative Features

- Imperative programs/ backtracking
    - Program :-
        - Member(X, [1,2,3,4]),
        - Write(X),
        - N1,
        - Fail
    - Program .
    - ?- program. %prints all solutions
- Fail: always fails
- ! Is cut operator: prevent other rules from matching

Arithmetic Operators

- int/float operators: +, -, *, /
- int operators: mod, //
- comparison operators: <, >, =<, >=.

Programming with Lists

- delete/3 to remove given element from a list ex delete([1,2,3], 2, X) X = [1,3].
- Algorithm: when X is select from [X|Ys], Ys results
- Permutations: permute/2 to find permutation of a given list

    - Permute([1,2,3],X) return X[1,2,3] upon backtracking X[1,3, 2], X[2,1,3], X[2,3,1] to X[3,2,1].
- Issue of Efficiency: rev/2 finds the reverse of given list rev([1,3, 2],X) succeed X = [3,2,1].

Tree traversal

- Binary tree represented by
    - Node/3facts: for internal nodes: node(a,b,c) means that a has b and c as children.
    - Left/1 facts: for leaves leaf(a) mean a is a leaf

Difference Lists

- Lists in prolog are singly-linked; hence we can access first element in constant time but need to scan entire list to get last element
- However we can use variables in data structure = to make lists open tailed
- When X = [1,2,3| Y], X is a list with 1,2,3 as its first element, followed by Y
    - Now if Y = [4|Z] then X = [1,2,3, 4| Z]
    - Z as pointing to the end of X
    - WE can now add an element to the end of X in constant time
    - Open-tailed lists are called difference lists in Prolog.
- Conventions:
    - difference list represented by 2 variables: one referring to entire list, another to its tail

Functional Programming with Common Lisp

1. Lisp

= List Processing Language
Basic datatype is list, programs themselves are lists

An ***element of a list*** can be either a ***list*** or an ***atom***. A list can also be empty.

```
()              ; The empty list.
(1 3 5 7)       ; A list of four elements, the numbers 1, 3, 5, and 7.
((1 2)(3 4))    ; A list of two elements, the list (1 2) and
                ; the list (3 4).
(((1 2)(3 4)))  ; A list of one element, the list ((1 2)(3 4)).
(a (b 1) 2)     ; A list with three elements: the symbol a,
                ; the list (b 1) and the number 2.
```

1.1 Anything in () is a function call

Ex (+ 1 2) evaluates 3 BUT ((+ 1 2)) error bc 3 is not a function
(quote(1 2 3)) short form '(1 2 3) is a list that contains +, 1, 2

1.2 Functional Programming

Evaluate functions
Avoid global state and mutable data
Higher-order first class functions
Closures and recursion
Lists and list processing

1.3 State

State of program = all of current variable and heap values
Imperative programs destructively modify existing state SET{x,y}
Functional programs yield new similar states over time SET_1{x} –> SET_2{x,y}

1.4 Functional-Style Advantages

- Tractable program semantics (procedures=fcts; formulate and assertions about code; more readable)
- Referential transparency (replace expression by value without changing result)
- No side-effects (fewer errors)

1.5 Functional-Style Disadvantages

- Efficiency (copying takes time)
- Compiler implementation (frequent memory allocation)
- New programming style = unfamiliar
- Not appropriate for every program

2. Basic Types

Atoms: symbols (words) || numbers || NIL (means false or empty list)
Lists: objects/expressions enclosed in ()
Ex: () the empty list or NIL = false
(3 (4 5 6) a b c)
((B (3))) : (a b c) is list of 3 items, ((a b c)) is list of 1 item which the item is a list of 3 items
Strings: sequence of char within double quotes
Ex: "this is a string"
Objects and other structures

2.1 Atoms

= number, symbols (words), NIL
Numbers are atoms Ex: 3, 8.9, 2/3
Symbols are atoms Ex: object whose name is a string
Value – symbols are "variables" they "bound" to other lisp objects Ex FOO bound 4.2
Function = fct name bound to fct definition
NIL is an atom it is an atom AND a list

2.2 T and NIL

= self-evaluating and ø be bound to anything

NIL : represents as symbol "false" as a list empty list "()"
T or ***ANYTHING non-NIL*** : represents "true"
(null <arg>) returns true if arg = nIL, false otherwise
Ex: returns T if arg NIL, else returns NIL
(listp <arg>) : returns true if arg = list (including NIL)
Ex: (listp NIL) = T

2.3 Variables

Values have types not variables

2.4 Evaluation of Basic Types

Numbers: self-evaluating ex 5 => 5
Symbols: evaluate hteir binding or value ex Color has value 3 then color => 3. Evaluation of an unbound symbol = error. Symbol have many facets to a value.
Strings: self-evaluation Ex "this is a string" => "this is a string"

2.4.1 preventing evaluation: quote

- ***(quote <arg>)***

prevents evaluation of its arg
its return value is <arg>

- ***Single quote mark '<arg>***

is equivalent to (quote <arg>)

Ex 'foo is the same as (quote foo)
'foo => foo

3. Functions

"function f is a mapping from each element in a set A to exactly one element in a set B.
f: A ➔ B "

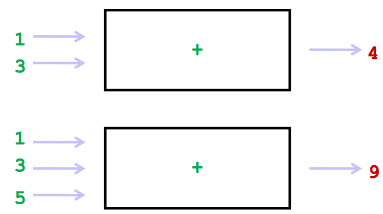
They're called using prefix notation

- Parentheses () surround the call
- *1st item*: function name || symbol
- *2nd & the rest*: parameters

(functionName arguments)

(+ 1 3) => 4

(+ 1 3 5) => 9



Ex:

```
(+ 1 2 3 4) ; Equivalent to infix (1 + 2 + 3 + 4). Returns 10.
(* 2 3 4)   ; Equivalent to infix (2 * 3 * 4). Returns 24.
(< 1 3 2)   ; Equivalent to (1 < 3 < 2). Returns false (NIL).
```

A function definition:
(defun *name* (*parameter* list)
body)

3.1 Functions Evaluation

function call has form – (<name> <arg1> <arg2> ...)
All args are evaluated
Fct definition of <name> is obtained
Fct is applied to args
Value's returned

Examples:

- `(+ 5.3 8)` => `13.3`
- `(+ (+ 4.5 (* 5.7 3)) (- 7 29) (/ 99 23))`
  => `3.9043465`
  - in infix notation this amounts to
    `(4.5+5.7*3)+(7-29)+(99/23)`
- if `X` is bound to `23`, then
  `(+ X 4)` => `27`

3.2 Functions and Control

Functions

- Built-in fct
- Defining fct
- Fct Evaluation and Special Forms *(defun, if)*

Control statements

- Conditional *(if, cond)*
- Repetition *(loops) - do*
- Sequence - *prog*

3.3 High-order Functions

Functions which do at least one of the following:

1. Take one or + fcts as their args
2. Return a fct

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

- ***mapcar***
  takes as args a fct and one or + lists & applies the fct to elements of list(s) in order.

Ex ; Multiplication applies to successive pairs.

```
> (mapcar #'* '(2 3) '(10 10))
(20 30)
```

- ***funcall***
  takes as args a fct and a list of args (ø require args to be packaged as list), and returns result of applying fct to elements of list.

Ex

```
> (funcall #'+ 1 3 4) ;    Equivalent to (+ 1 3 4).
        8
```

- ***apply***
  works like funcall, but requires that last arg is a list.

Ex

```
> (apply #'+ 3 4 '(1 3 4))
        15
```

3.4 Anonymous Functions

It's one that is defined (and called) without being bound to an identifier.
They aren't stored in memory.
Syntax: (lambda (formal parameter list) (body))

3.5 Recursive functions

Each recursive case consists of:

1. Splitting data into smaller pieces
2. Handling pieces with calls to current method
3. Combining results into a single result

Ex f : N → lists(N)

f (0) = ⟨ 0 ⟩.
f(n)=cons(n,f(n−1)), for n>0.

Ex sum

```
 (defun sum (lst)
      (cond ((null lst) 0)
             (t (+ (car lst) (sum (cdr lst))))))
```

Ex find last element

```
      (defun last2 (lst)
             (cond ((null lst) nil)
                   ((null (cdr lst)) (car lst))
                   (t (last2 (cdr lst)))))
```

Ex reversing a list

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

4. Basic List Processing Functions

1. ***List***
   creates a list comprised of its arguments
   Ex (list 'x 'y 'z)=>(X Y Z)
2. ***Car (or first)***
   returns the first element of a list
   Ex (car (list 'x 'y)) => X
3. ***Cdr (or rest***)
   Everything but first element
   Ex (cdr '(a b c))=>(B C)
4. ***Cons***
   creates a list by adding element as head of existing list
   Ex cons (list 'x 'y) (list 'x 'y)) => ((X Y) X Y)
5. **Append**

Takes any number of lists as args
Returns them appended together
Ex (append '(a b c) '(d e f)) => (A B C D E F)

6. **Equal**
   Takes 2 args
   Returns T if they are structurally equal or equal value

ATTENTION

EQL: return true if its arguments point to the same object
Ex: (eql 1 1) is true (eql 1 1.0) is false
EQUAL: returns true if its arguments have the same value.

4.1 Predicate Functions listp and null

Predicate: function whose return value is intended to be interpreted as T/F

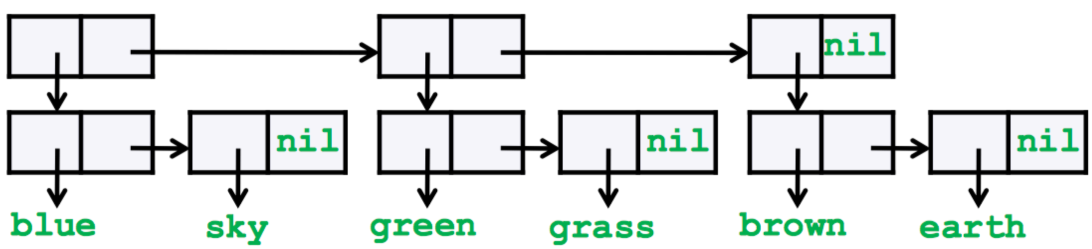
Listp

- takes one parameter
- it returns
  • T if the arg is a list
  • NIL otherwise

Null

- takes one parameter – it returns
  • T if the parameter is the empty list
  • NIL otherwise
- Note that null returns T if the parameter is ()!

4.2 Sublists



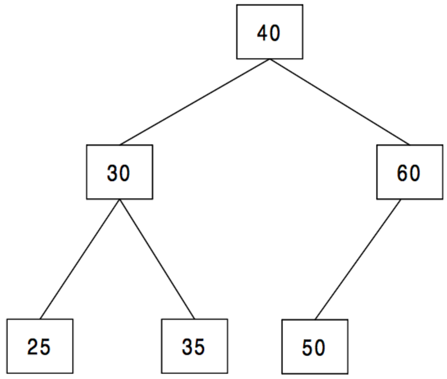
5. Cons Cell and Lists as Trees

Binary tree: car -left subtree & cdr -right subtree

We can use a list to represent a non-empty tree as ***⟨atom, ⟨l – list⟩, ⟨r – list⟩⟩***

atom = root

Ex   ’(40 (30 (25 () ())(35 () ()))(60 (50 () ())()))



6. Variables

6.1 Quote

takes 1 param & entire expression evaluates to param

Ex (quote (a b c))=>(A B C)   OR   '(a b c) => (A B C)

Doesn’t work under eager evaluation

Ex: (+ 1 2) is a program

'(+ 1 2) is data (list of 3 elements)

(+ 2 (+ 1 2)) is 5

(+ 2 '(+ 1 2))is an error => ø evaluate +1 2

6.2 Variable declarations

Global and special variables

- `(defvar …)`
- `(defparameter …)`
- `(defconstant …)`
- `(setq var2 (list 4 5))`
- `(setf …)`

Local variables

- `(let ((x 10)) …)`
- `(let* ((x 99) …)`

6.3 Global variables

Var is ***global*** if it is visible ***everywhere*** as opposed to a ***local*** variable which is visible ***only within code block*** in which it’s defined.

Setq = global var
Setf = local & global

Declared with defvar and defparameter
(defvar <var name> <init value> <documentation>)
Assign values with setf
(setf <place> <new value>)

7. Expressions and functions

Expressions are written as ***lists, using prefix notation***.
Prefix notation: form of notation for logic, arithmetic, and algebra. It places operators to the left of their operands.

Ex expression 14 − (2 × 3) is written as (− 14 (× 2 3))

7.1 Arity of functions

Arity: describe the number of arguments or operands that a function takes.
Unary fct = arity 1 & binary fct = arity 2, etc.…

7.2 Prohibiting expression evaluation

Quote will stop the expression evaluation
Ex: (/ (* 2 6) 3) returns 4
BUT ’(/ (* 2 6) 3) returns (/(*2 6) 3)

7.3 Boolean Operations

Lisp supports Boolean logic with operators and, or, and not

7.3.1 OR operator

Evaluates its subexpressions from ***left to right*** and ***stops*** immediately if any subexpression is ***T***

Ex
```
> (let ((x 5))
(or (< x 2) (> x 3)))
T
```

7.3.2 AND operator

Evaluates its subexpressions from ***left to right*** and ***stops*** immediately if any subexpression is ***false/NIL***.

Ex
```
> (let ((x 5))
 (and (< x 7) (< x 3)))
NIL
```

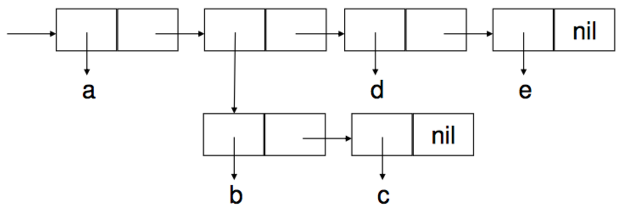
8. Constructing lists

3 fcts to create list:

1. cons 2. list 3. append

8.1 Fct cons

A list in Lisp is ***singly-linked*** where each node = pair of 2 pointers (first one pointing to data element & 2nd pointing to tail of list with last node's 2nd pointer pointing to empty list)



8.2 Fct list

Lists can be created directly with list function, which takes any number of args, and it returns list composed of these args.

```
(list 1 2 'a 3)               ; Returns (1 2 A 3).
(list 1 '(2 3) 4)             ; Returns (1 (2 3) 4).
(list '(+ 2 1) (+ 2 1))       ; Returns ((+ 2 1) 3).
(list 1 2 3 (list 'a 'b 4) 5) ; Returns (1 2 3 (a b 4) 5).
```

8.3 Fct append

It takes any number of list args and it returns list which is concatenation of its args:
Ex

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

append expects as its arguments only lists.

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

9. Accessing a List

Only access either the head of a list, or the tail of a list.
Operation car takes a list as an argument and returns the head of the list.
Operation cdr takes a list as an argument and returns the tail of the list.
Ex (car (cdr '(1 (3 5) (7 11)))) Returns (3 5)

10. Control flow

Single selection

> **if**

```
( if    testExpression
        thenExpression )
( if    testExpression
        thenExpression
        elseExpression )
```

Multiple Selection
Can be formed with ***cond*** which ***contains list of clauses*** where ***each*** clause ***contains*** 2 expressions, called ***condition*** and ***answer***.

```
Ex: (cond (question answer)
        ...
        (else answer ) ; Optional
         )
```

Conditions evaluated sequentially. Can use t(true) instead of else.

11. Binding

11.1 Variables and binding

Binding = mechanism for implementing lexical scope for variables.

```
Ex      ( let
                ( (binding1)
                (binding2 )
                ... )
                (expression) )
```

where (binding$_n$) is of the form (variable$_n$ value)

values are computed and bindings are done in parallel.

```
let ((x 2) (y 3))
    (+ x y))            Returns 5
```

11.2 context and nested binding

Inner binding for variable shadows outer binding and region where variable binding is visible is called ***its scope***.

let* is functionally equivalent to a series of nested lets

```
Ex      (let* ((x 10)
                 (y (* 2 x))) ; Not legal for let.
                (* x y))
        ; Returns 200
```

12. Search a List

12.1 Bubble Sort

is-sortedp which returns True or False on whether or not its list argument is sorted

```
(defun is-sortedp (lst)
  (cond ((or (null lst) (null (cdr lst))) t)
    ((< (car lst) (car (cdr lst))) (is-sortedp (cdr lst)))
    (t nil)))
```

bubble-sort

```
(defun bubble-sort (lst)
  (cond ((or (null lst) (null (cdr lst))) lst)
        ((is-sortedp  lst) lst)
        (t (bubble-sort (bubble lst)))))
```

12.2 Linear Search

If x appears in L, then we would like to return its position in the list.

```
(defun search (lst elt pos)
  (if (equal (car lst) elt)
    pos
    (search (cdr lst) elt (+ 1 pos))))

(defun linear-search (lst elt)
  (search lst elt 1))
```

12.3 binary Search

```
(defun binary-search (lst elt)
 (cond ((null lst) nil)
   ((= (car lst) elt) t)
   ((< elt (car lst)) (binary-search (car (cdr lst)) elt))
   ((> elt (car lst))
       (binary-search (car (cdr (cdr lst))) elt))))
```