Presentations

Presenting teams should concentrate on explaining their solutions technically and how they conducted their problem solving (ie: you should not spend time animating power-point slides, etc).

You are encouraged to present two or more solutions to a problem when there are different ways that you can deal with the problem. For example... you may find some problems can be dealt with using a simple pattern matching expression. In this case we would expect you to explore solutions which do not use the matcher as well as those that do.

presentation problem 1.1a

Produce a function which takes as its 2nd argument a list of lists where each sublist is of the form (name associated-value), and as its 1st argument a word. If that word occurs as a name somewhere in its list argument then function should return the associated value. If the word is not found then the function returns nil. (NB: at least one approach to this should present a recursive solution).

eg:
```
    (findit 'bus '( (banana yellow)
                    (bus    red   )
                    (frog   green )
                  ))
==> RED


    (findit 'egg '( (banana yellow)
                    (bus    red   )
                    (frog   green )
                   ))
==> NIL
```

presentation problem 1.1b

Write a function to take a list of data describing a shopping spree. Each entry in the list is of the form `(item number cost)`, your function should return the total value of all shopping (costs are in pounds).

eg:
```
    (total-cost
       '( (banana 5  0.70)
          (egg    6  0.30)
          (crisps 2  0.50)
          (wine   3  4.99)
        ))
==> 22.27
```

presentation problem 1.1c

Data describing object locations is held in a series of lists of the form…
(object-id category super-category location)

eg:
```
    (defparameter *obj-data*
        `((apple#3  apple  fruit  kitchen)
          (mango#5  mango  fruit  kitchen)
          (tom      cat    agent  hallway)
          (jerry    mouse  agent  bedroom)
           -etc-
         ))
```

Write a function which takes three arguments (i) a super-category (ii) a location name (iii) a data set and returns a nil/non-nil value (indicating false/true) indicating whether an instance of the super-category can be found in the named location (NB: your solution will be better if the non-nil values it returns are useful).

presentation problem 1.1d

A data-structure contains transport information...

```
    (defparameter *transport*
        `((train newcastle durham darlington)
          (distance newcastle middlesbrough 35)
          (distance middlesbrough saltburn 10)
          (train darlington middlesbrough saltburn)
          (bus   newcastle  middlesbrough)
          (distance middlesbrough newcastle 35)
         -etc-
        ))
```

Write a function which takes this kind of structure and also the names of 2 places as its 3 arguments and returns the distance between the two places if (and only if) there is a direct link between them.

There is a direct link between A and B only if there is a tuple of the form...
(distance A B ?) where ? is a numeric value

Note that distance tuples are one-way only so (distance B A ?) does not imply (distance A B ?)

presentation problem 1.2

Write a function, splice-out, which takes 3 arguments: a list and two indexes. The indexes mark start and end points within the list (lists, by convention, are indexed from 0 by CL programmers). The function should return those elements in the list that fall between the two indexes.
eg:
```
        (splice-out '(a b c d e) 1 3)  ==>  (b c d)
```

You should be clear about the way your function will behave in situations where the indexes are outside the bounds of the list.

presentation problem 1.3

Write a function which takes two lists of numbers (both assumed to be in ascending order) and merges them into a single list (also in ascending order).
eg:

```
        (merge2 '(2 5 6 12) '(1 4 5 11 15))
        ==> (1 2 4 5 5 6 11 12 15)
```

presentation problem 1.4

Develop a function called find-indexes which takes 2 arguments, an item & a list. You may assume that the list will always be a flat list. The function is to return the position of the item in the list, if it is found.
eg:

```
    ==
    == (find-indexes 'X '(a b c X d e))
    (3)
    ==
```

The function must be able to deal with multiple occurrences of the search item, returning a list of relevant indexes.
eg:

```
    ==
    == (find-indexes 'X '(a b X c X d e))
    (2 4)
    ==
```

presentation problem 1.5

Data describing object locations is held in a series of lists of the form…
(object-id category super-category location)

eg:

```
    (defparameter *obj-data*
        '((apple#3  apple  fruit  kitchen)
          (mango#5  mango  fruit  kitchen)
          (tom      cat    agent  hallway)
          (jerry    mouse  agent  bedroom)
           -etc-
         ))
```

Write a function which takes this type of data-structure and a super-category as its two arguments and returns a list of locations where the super-category items can be found. You may also consider writing the function so it returns more useful results.

presentation problem 2.1

Write a function called count+nodes, which takes a nested list as its only argument (ie: a tree). count+nodes should return a count of the number of positive numbers in the tree.
eg:
```
==
== (count+nodes '(1 (-2 dog 17 (4) cat) -8 (rat -6 13) mango))
4
==
```

presentation problem 2.2

Develop a function called bounds which takes a nested list of numbers as its only argument (ie: a tree). bounds should return a list (or cons-pair) of the largest & smallest value in the tree.
eg:
```
==
== (bounds '(1 (-2 17 (4)) -8 (-6 13)))
(-8 17)
==
```

presentation problem 2.3

Write a function which takes one arg & returns a count of the number of cons-pairs in the Lisp representation of its arg. Look carefully at the following examples...

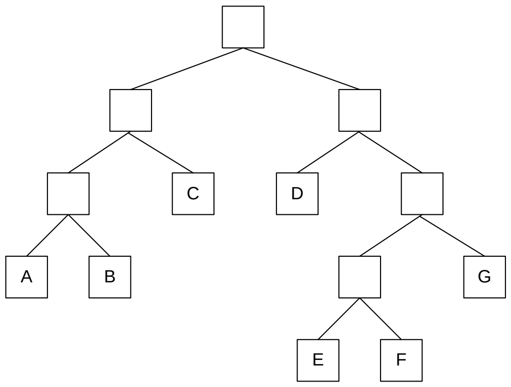
```
(count-cons nil)      →  0
(count-cons 'apple)   →  0
(count-cons '(a.b))   →  1
(count-cons '(a b))   →  2
(count-cons '(cat bat rat))          →  3
(count-cons '(cat (bat mat) rat))    →  5
(count-cons '((cat rat) (bat mat)))  →  6
```

presentation problem 2.4

Write a function called nested-average, which takes a nested list of numbers as its only argument (ie: a tree). nested-average should return the average of all numbers in the tree.
eg:
```
==
== (nested-average ' (10 ((30 1) 20) (8 (5 (50 7)) 9) 40))
18
==
```

presentation problem 2.5

Strict binary trees have two branches from each non-terminal node: a left branch & a right. We can conveniently represent binary trees using a dotted notation, eg:



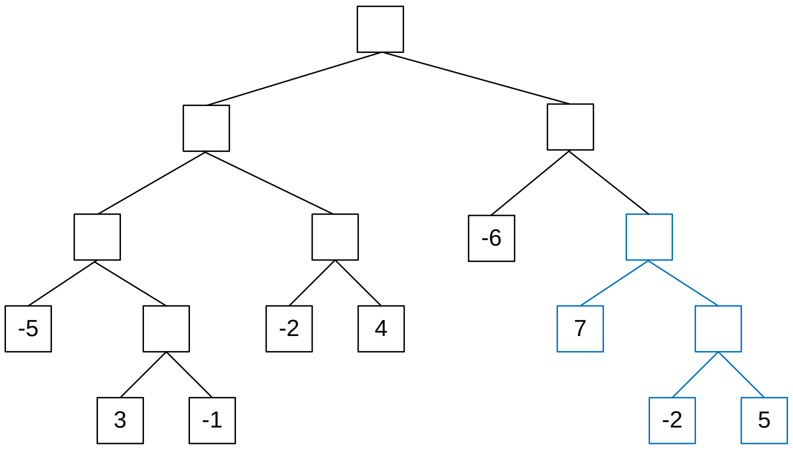
...can be represented as:

```
    '(((a . b). c).(d .((e . f). g)))
```

but note that Lisp does not print in dotted notation so this list will be printed as...

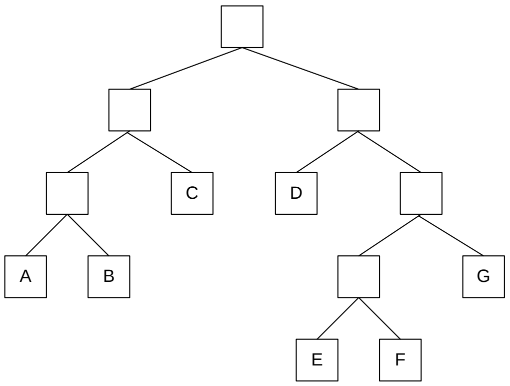
```
    (((a . b) . c) d (e . f) . g)
```

Write a function maximum-tree-sum which takes a binary tree containing positive & negative numbers & returns the sum of numbers in the subtree which contains the largest of these sums. For example, with the tree below: the maximum-tree-sum is 10, summed from the blue subtree.



presentation problem 2.6

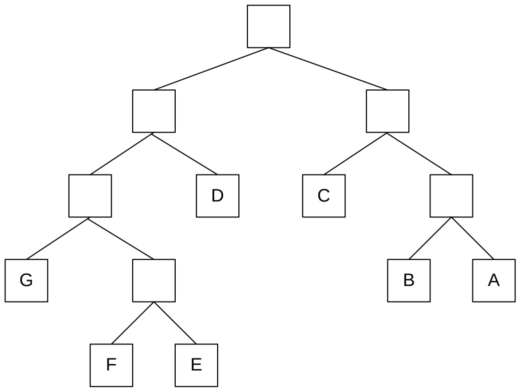
Strict binary trees have two branches from each non-terminal node: a left branch & a right. We can conveniently represent binary trees using a dotted notation, eg:



...can be represented as:

```
    '(((a . b). c).(d .((e . f). g)))
```

Write a function *spin,* which takes a binary tree as its argument & returns a tree formed by rotating each of the non-terminal nodes in the original tree. The tree above would look like this...



Note that Lisp does not print in dotted notation so...

```
    '(((a . b). c).(d .((e . f). g)))
```

...will be printed as...

```
    (((a . b) . c) d (e . f) . g)
```

presentation problem 3.1

This presentation is dealing with trees. While trees have similar structures their representations in lists can vary a little. For this problem the representation of a tree is as follows...

Trees are made up of nodes, some are terminal (ie: at the end of branches & have nothing hanging off them) others are non-terminal nodes & these have branches from hanging them. In some trees all nodes are labelled, in others only the terminal nodes are labelled. The tree below has only labels only for its terminal nodes. Non-terminal nodes are marked * to make them more visible.

```
       *
     /   \
    /     \
   *       *
  / \     / \
cat  *  dog  rat
    / \
  bat  gnat
```

This tree could be represented as:

```
        '( (cat (bat gnat)) (dog rat) )
```

The maximum depth of a tree is measured by the number of non-terminal nodes that have to be traversed to reach the terminal node furthest from the root node (the node at the top level).
Your task...

write a function "max-tree-depth" which takes a tree as its one and only argument and returns the maximum depth of the tree. Tree depth starts from 0 and the max-tree-depth of the tree...

```
      '(a ( (b (c d)) e) (f g) h)
```

should either be 3 or 4 depending on your interpretation of tree structure.

presentation problem 3.2

This presentation is dealing with trees. While trees have similar structures their representations in lists can vary a little. For this problem the representation of a tree is as follows...

Trees are made up of nodes, some are terminal (ie: at the end of branches & have nothing hanging off them) others are non-terminal nodes & these have branches from hanging them. In some trees all nodes are labelled, in others only the terminal nodes are labelled. The tree below has only labels only for its terminal nodes. Non-terminal nodes are marked * to make them more visible.

```
           *
         /   \
        /     \
       *       *
      / \     / \
    cat  *  dog  rat
        / \
      bat  gnat
```

This tree could be represented as:

```
        '( (cat (bat gnat)) (dog rat) )
```

The maximum breadth (also known as maximum branching factor) of the tree is measured by finding the greatest number of branches that occur from any non-terminal node.

Your task...

write a function "max-breadth" which takes a tree as its one and only argument and returns the maximum breadth of the tree, eg:

```
==
== (max-breadth '(a ( (b (c d)) e) (f g (h i) j))
== 4
==
```

presentation problem 3.3

This presentation is dealing with trees. While trees have similar structures there representations in lists can vary a little. This problem deals with an expression tree which is represented as a nested list where all head items are mathematical symbols and all other terminal nodes are numbers.

Produce a function which takes an expression tree as its argument and returns that tree with the relevant calculated values in place of the operators.

Eg:

```
     *        21
    / \  =>  / \
   3   7    3   7
```

So:
```
(evaltree '(* (+ 5 (* 3 7)) (- 6 8)) )
==> (-52 (26 5 (21 3 7)) (-2 6 8))
```

HINT : apply is a CL function which applies other functions, eg:
(apply '* '(3 4)) ==> 12

presentation problem 3.4

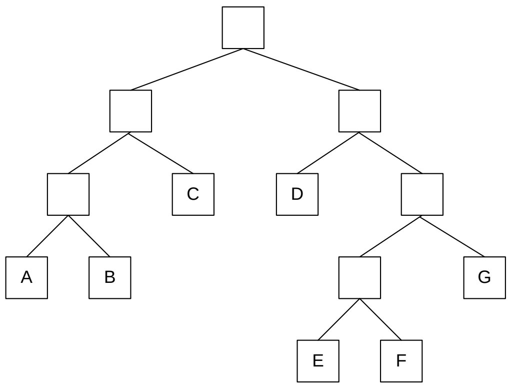
This presentation deals with generalised trees – represented as nested lists.

Write a function which takes a tree as its argument and returns a nil/non-nil value indicating whether the tree contains only unique nodes (ie: there are no duplicated nodes in the tree).

HINTS :
there are various solutions to this problem, in one type of solution you may find it useful to break one of our normal programming rules and use a conditional form (if, unless, etc) *inside* an action clause of a cond statement. You may also find it useful (in a couple of different types of solution) to use a *let* expression to set local variables.

presentation problem 3.5

Strict binary trees have two branches from each non-terminal node: a left branch & a right. We can conveniently represent binary trees using a dotted notation, eg:



...can be represented as:

```
    '(((a . b). c).(d .((e . f). g)))
```

Write a function *tree-path* which takes a symbol and a binary tree as its two arguments & returns a sequence of left/right braches which trace a path to where the symbol can be found in the tree, eg: the path to 'f in the tree above is (right right left right).

If the symbol cannot be found in the tree your function should return nil.

NB: you should also consider how the function could operate if the tree is allowed to contain duplicate entries.

presentation 4.1

A set of tuples holds information about a house which includes tuples describing which rooms connect to which other rooms, eg:

```
(connects door5 bedroom hall)
(connects door5 hall bedroom)
(connects blue-door kitchen hall)
(locks key6 blue-door)
(location stove kitchen)
...etc...
```

NB: (i) not all doors open in both directions so do not assume that (connects d a b) implies (connects d b a) and (ii) as shown above not all of the tuples describe "connects" relationships.

Write a function which takes 3 arguments as follows...

tuples	- a set of tuples like those above
start	- the name of a room to start in
goal	- the name of a room to end in

Your function should return either...

(i) nil – if there is no way of getting from the start room to the goal room (ie: they do not directly or indirectly connect to each other)

(ii) non-nil – if there is some way (perhaps via other rooms) to get from the start to the goal. A good solution will return some non-nil value that may be useful.

HINT: check out the examples section of the matcher documentation under downloads on www.agent-domain.org – particularly where it examines implementing a grandparent relation.

presentation 4.2

A set of tuples holds information about distances between towns/villages, eg:

```
(distance newcastle middlesbrough 35)
(distance middlesbrough saltburn 10)
(distance middlesbrough newcastle 35)
(distance newcastle durham 15)
...etc...
```

NB: (i) do not assume relationships are bi-directional so do not assume that distance from A to B is the same as the distance from B to A (this actually makes things easier). (ii) if there is no tuple for getting from A to B then it implies there is no direct route from A to B.

Write a function which takes 3 arguments as follows...

tuples	- a set of tuples like those above
start	- the name of a town to start in
goal	- the name of a town to end in

Your function should return either...

(i) the distance between towns – if there is some way (perhaps via other towns) to get from the start to the goal.

(ii) nil or some other sensible value if there is no way to get from the start to the goal.

HINT: there are various ways to implement this, if you want to use the matcher then check out the examples section of the matcher documentation under downloads on www.agent-domain.org – particularly where it examines implementing a grandparent relation.

presentation 4.3

A set of tuples holds hierarchical species information about distances between animals etc, eg:

```
     (has bird feathers)
     (color budgie yellow)
     (eats budgie seed)
     (color tweetie green)
     (isa tweetie budgie)
     (isa budgie bird)
     ...etc...
```

NB: the "isa" relationship is of special importance – it defines an upwards link in the species hierarchy.

Write a function called "inherit" which takes 3 arguments as follows...

tuples	- a set of tuples like those above
object	- the name of an animal or species (tweetie, budgie, etc)
relation	- the name of a relation (color, has, etc)

Your function should return either...

(i) the value of the relation for the named animal/species either directly or inherited from its species/super-species (see examples below)

(ii) nil

Examples

```
     (inherit tuples 'tweetie 'heart-rate) => nil
     (inherit tuples 'tweetie 'color)      => green
     (inherit tuples 'tweetie 'eats)       => seeds
     (inherit tuples 'tweetie 'has)        => feathers
```

HINT: there are various ways to implement this, if you want to use the matcher then check out the examples section of the matcher documentation under downloads on www.agent-domain.org – particularly where it examines implementing a grandparent relation.

presentation 4.4

A set of tuples holds information about a house which includes tuples describing which rooms connect to which other rooms via which doors, the current status of doors (locked or unlocked) and which keys operate which doors, eg:

```
(connects door5 bedroom1 landing)
(connects door5 landing bedroom1)
(connects blue-door kitchen hall)
(locks key6 blue-door)
(status green-door unlocked)
(status blue-door locked)
(status door5 locked)
(locks bedroom-key door5)
(connects green-door hall stairs)
(connects <no-door> stairs landing)
(status <no-door> open)
...etc...
```

Write a function which takes 2 arguments as follows...

tuples	- a set of tuples like those above
route	- a sequence of rooms eg: (kitchen hall stairs landing bedroom1)

Your function should return a list of keys required to make the journey through the rooms (ie: those which unlock any locked doors which need passing through).

HINT: there are various ways to implement this, if you want to use the matcher then check out the examples section of the matcher documentation under downloads on www.agent-domain.org.

presentation 5.1

In some of the earlier lectures & tutorials we started to develop a simple pattern matcher. We got as far as a kind of equals function which also allowed a single wild card symbol. This presentation problem requires you to build a more developed matcher.

You should build a matcher that can operate on nested lists and manage a primitive form of match-variable. Look at the following examples of using a matcher (called ***mm***) – your matcher function should produce the same output (though the order of items in the output is not important).

```
(mm '(a b c) '(a b c d))  →  nil

(mm '(a b c d) '(a b c))  →  nil

(mm '(a (b c) d) '(a (b c) d))  →  ((<pattern> (a (b c) d)))

(mm '(a = d) '(a (b c) d))  →  ((<pattern> (a = d)))

(mm '(a ?x d) '(a (b c) d))  →  ((?x (b c)) (<pattern> (a ?x d)))

(mm '(a (b ?x) d ?x) '(a (b c) d c))
       →  ((?x c) (<pattern> (a (b ?x) d ?x)))

(mm '(a (b ?x) d ?x) '(a (b c) d e))  →  nil

(mm '(a (b ?x) d ?y) '(a (b c) d e))
       →  ((?y e) (?x c) (<pattern> (a (b ?x) d ?y)))
```

A couple of things to note...

1. you need to check whether a symbol starts with a "?" to determine whether it is a match variable – see below for a function that can make a test function for you (but note that you will have to call it using funcall as in the example)
2. you may want to use local variables in your function – check out the use of ***let*** in the Lisp documentation for this, here is a quick example of let

```
(let ((x 5)                  ; local x = 5
      (y (* 10 (+ 7 6)))     ; local y = 130
      )                      ; close variable declaration
  (+ x y)                    ; body of let
  )                          ; close let
                             ; variables do not exist outside of let
  →  135
```

presentation 5.2

Rules are used in various different ways in AI systems. Language processors use rules (grammars) like...

sentence $\rightarrow$ noun-phrase verb-phrase
noun-phrase $\rightarrow$ determiner noun
verb-phrase $\rightarrow$ verb noun-phrase

...etc, to pull English sentences apart. Language generators use the rules in a different way to generate English sentences.

Your task is to randomly generate sentences. Start by using a simple grammar like the one below...

```
(defparameter *rules1*
  '((!S (!NP !VP))
    (!NP (!D !N)
         (!D !Adj !N))
    (!VP (!V !NP))
    (!D the a)
    (!N cat rat bat)
    (!V ate chased)
    (!Adj big blue small pink black)
    ))
```

The rules read as follows...

```
(!S (!NP !VP))
```

...reads "a **S**entence is a **N**oun**P**hrase followed by a **V**erb**P**hrase"

```
(!NP (!D !N)
     (!D !Adj !N))
```

...reads "a **N**oun**P**hrase is a **D**eterminer followed by a **N**oun" or...
" a **N**oun**P**hrase is a **D**eterminer followed by an **Adj**ective then a **N**oun"

```
(!N cat rat bat)
```

...reads "**N**ouns are *cat*, *rat* & *bat*".

Your task is to write a function which randomly generates a string of words from the grammar each time it is called. Eg...

```
> (gen *rules1* '!S)  →  (the rat chased a rat)
> (gen *rules1* '!S)  →  (a bat ate the pink rat)
> (gen *rules1* '!S)  →  (a blue cat ate a big cat)
```

Note that, since the output is randomly generated, you will get different sentences produced but they should obey the rules of the grammar. Note also that *non-terminal* symbols (like !NP) which expand into other symbols are prefixed with "!" – this is to make it easier to detect them. Your strategy should be to expand non-terminal symbols until you get to words. When you have a choice, choose expansions randomly (see the ***one-of*** function below).

Once you have written the function for the grammar above, test it with this one...

```
(defparameter *rules2*
  '((!S (!NP !VP)
        (!NP !VP and !VP)
        (!S and !S))
    (!NP (!D !N)
         (!D !AdjP !N))
    (!AdjP !Adj
           (!Adj !AdjP))
    (!VP (!V !NP)
        (was !V by !NP))
    (!D the a)
    (!N cat rat bat)
    (!V ate chased)
    (!Adj big blue small pink black)
    ))

> (pprint (gen *rules2* '!S))
(the black bat chased the pink blue bat and was chased by a
     small small pink rat)

> (pprint (gen *rules2* '!S))
(the rat was chased by a rat and chased the small blue blue
     rat and a small bat was chased by the bat and chased
     the black rat)

> (pprint (gen *rules2* '!S))
(a cat was ate by a cat and ate the blue rat)

> (pprint (gen *rules2* '!S))
(the bat was chased by a small blue blue rat and ate the
     bat)
```

additional information for presentations 5.1 & 5.2

The function to make prefix checkers...

```
(defun make-prefix-test (prefix)
  (let* ((pstr (string prefix))
         (plen (length pstr))
         )
    #'(lambda (sym)
        (and (symbolp sym)
             (let* ((symstr (string sym))
                    (symlen (length symstr))
                    )
               (and (> symlen plen)
                    (string= pstr (subseq symstr 0 plen))
                    ))
             ))
    ))


> ;; make function called "?-p" to check symbols starting with "?"

>(setf ?-p (make-prefix-test '?))
#<Closure (:internal make-prefix-test 0) @ #x213564a2>

>(funcall ?-p '?spud)  →  t
>(funcall ?-p 'banana)  →  nil
>(funcall ?-p '?)  →  nil
```

A function to return a randomly chosen element of a list...

```
(defun one-of (lis)
  (elt lis (random (length lis))))

> (one-of '(spam egg chips))  →  chips
> (one-of '(spam egg chips))  →  egg
> (one-of '(spam egg chips))  →  egg
> (one-of '(spam egg chips))  →  chips
> (one-of '(spam egg chips))  →  spam
```