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.

Note also that in Clojure (more than other Lisp dialects) we tend to use lists only when they provide us with advantages, preferring to use vectors, maps & sets when these are more appropriate. This accepted, many of the problems described below are specified using lists this is (i) because we want you to get familiar with list manipulation and (ii) we expect you to consider other possible data structures as part of your problem analysis.

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/false
```

Consider which other Clojure data types may be appropriate to structure your problem data.

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)
        ))
==> 21.27
```

Consider which other Clojure data types may be appropriate to structure your problem data.

presentation problem 1.1b2

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 sub-totals (in a map) and 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)
        ))

==> { banana 3.5
      crisps 1
      wine   14.97
      total  21.27
      egg    1.8  }
```

Consider which other Clojure data types may be appropriate to structure your problem data.

presentation problem 1.1c

Data describing object locations is held as tuples of the form…
(object-id category super-category location)

eg:

```
(def 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 eg: (present? 'fruit 'kitchen obj-data) and returns a nil/false or 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).

Consider which other Clojure data types may be appropriate to structure your problem data.

presentation problem 1.1d

A data-structure contains transport information...

```
(def 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 *n*) where *n* is a numeric value

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

Consider which other Clojure data types may be appropriate to structure your problem data.

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). 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 set of tuples of the form…
(object-id category super-category location)

eg:

```
(def 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 proportion+nodes, which takes a nested list as its only argument (ie: a tree). count+nodes should return the proportion of nodes numbers in the tree which are positive numbers.
eg:

```
==
== (proportion+nodes '(1 (-2 dog 17 (4) cat) -8 (rat -6 13) mango))
4/11
==
```

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 the largest & smallest value in the tree.
eg:

```
==
== (bounds '(1 (-2 17 (4)) -8 (-6 13)))
(-8 17)
==
```

problem 2.2b

as problem 2.2 but the tree may contain mixed values (not only numbers).

presentation problem 2.3

A function *split* takes a tree containing mixed data (symbols & numbers). Split returns a list of two structures, one containing all the symbols, the other containing all the numbers. NB: the ordering of data in the returned structure is not important.
eg:

```
==
== (split '(1 (-2 dog 17 (4) cat) -8 (rat -6 13) mango))
{:nums (1 -2 17 4 -8 -6 13) :syms (dog cat rat mango)}
```

problem 2.3b

Similar to problem 2.3 but split now takes an argument which identifies how to split the data
eg:

```
==
== (split '(1 (-2 "string" dog [this is a vector] 17 (4) cat) -8
              [and another  vector] (rat -6 13) "blah" mango)
           {:nums {:test number?}, :vects {:test vector?}
            :syms {:test symbol?}})
{:nums {:test number?, :data (1 -2 17 4 -8 -6 13)}
 :syms {:test symbol?, :data (dog cat rat mango)}
 :vects {:test vector?,
         :data ([this is a vector][and another vector])}
 :other {:data ("string" "blah")}}
```

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
```

problem 2.4b

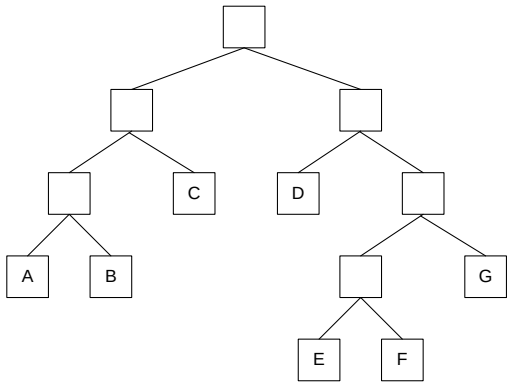
Similar to 2.4 but the new function (called "stats") returns the smallest, largest, count and average profile of numbers in a tree. Values to be returned in a map.

problem 2.4c

As above but takes a sequence of comparators (like <, >=, or user defined) to determine which stats are returned.

presentation problem 2.5

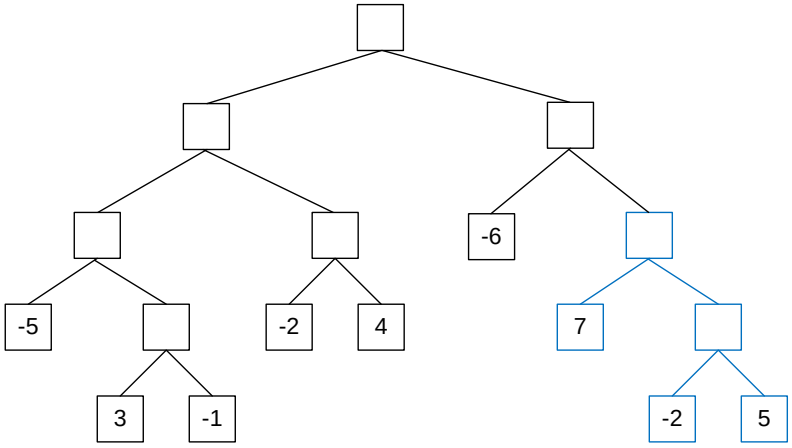
Strict binary trees have two branches from each non-terminal node: a left branch & a right. We can conveniently represent binary trees using lists or vectors, eg:



...can be represented 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.

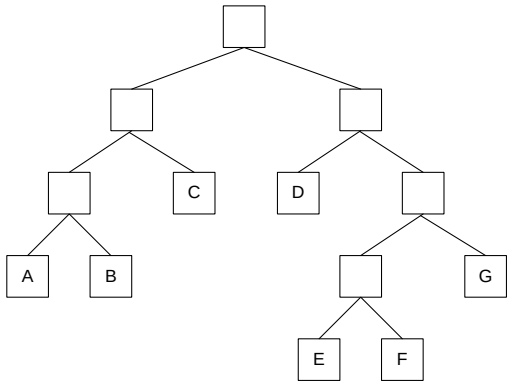


problem 2.5b

Extend 2.5 so it returns (i) the maximum tree-sum (ii) the minimum tree-sum (iii) a sequence of all zero-sum subtrees.

presentation problem 2.6

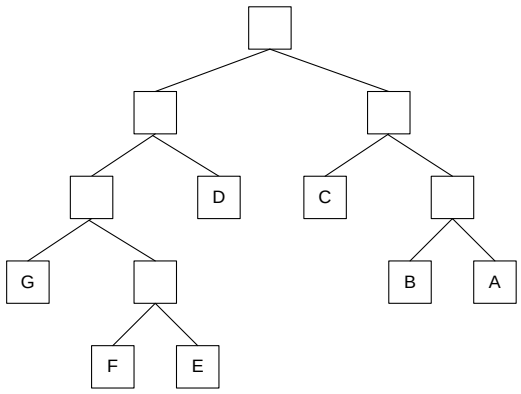
Strict binary trees have two branches from each non-terminal node: a left branch & a right branch. We can conveniently represent binary trees using lists or vectors, eg:



...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...



2.6b – spin & palindrome

As problem 2.6 but also reports on any trees it finds which are palindromes.

2.6c – spin power-set

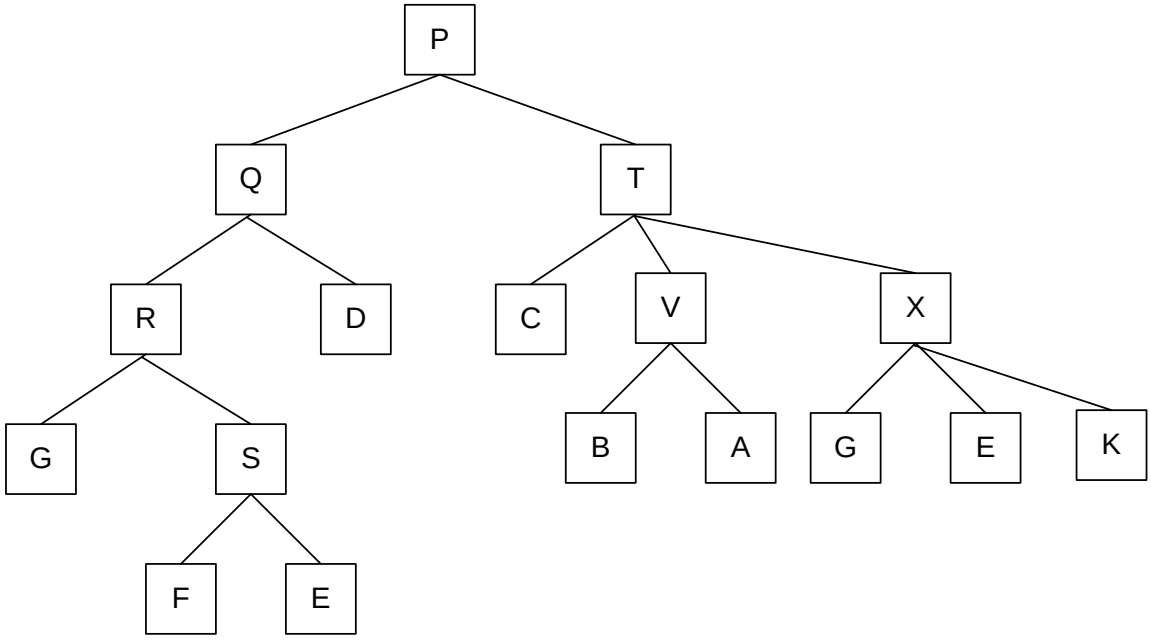
A spin-power-set (our term invented here) is a power-set of all trees where any of its subtrees may be spun/rotated or not. So: `[c [b a]]`, `[[b a] c]`, `[c [a b]]`, `[[a b] c]` are all part of the same set but `[a, [b c]]` is not.

Write a function to return the spin-power-set of a tree.

2.6c – order-if

order-if is like spin (see 2.6) but it takes a slightly different tree structure and a comparator to determine how & when trees are reorganised.

In this case, all nodes have values, even non-terminal nodes and trees can have varying numbers of branches so this would be a tree...



...represented as a vector like this...

[P [Q [R G [S F E]] D] [T C [V B A] [X G E K]]]

order-if takes a comparator as an argument & reorders nodes at each level according to the comparator. So if we used an alphabetic-less-than on the node X it would be reordered as follows...

from: [X G E K] to: [X E G K]

...and the node Q...

from: [Q [R G [S F E]] D] to: [Q D [R G [S E F]]]

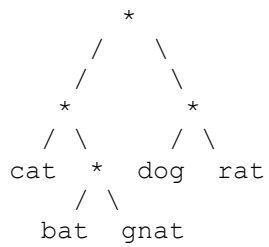
problem 2.6d

as above but produces a map of the following: the new tree; no. of subtrees unchanged by reordering; approximate volume of the tree unaffected by re-ordering.

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.



This tree could be represented using vectors as:

```
'[[cat [bat gnat]] [dog rat]]
```

or using lists:

```
'((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...

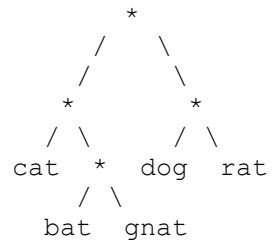
```
'(a ( (b (c d)) e) (f g) h)
```

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.



This tree could be represented using vectors as:

```
'[[cat [bat gnat]] [dog rat]]
```

or using lists:

```
'((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
==
```

3.2b – breadth&depth

This returns both the results from problem 3.1 and the results from problem 3.2, ie: given a tree returns both its max-depth and max-breadth. This function should also return the most minimally populated sub-tree where sub-tree population is determined by

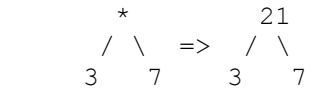
b * d / no.-of-nodes

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:



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

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

problem 3.3b

extend problem 3.3 so it also deals with other types of functions & data, eg…

```
(first (rest (map inc (quote 5 7 9 11))))
(8 ((8 10 12) ((6 8 10 12) (5 7 9 11))))
```

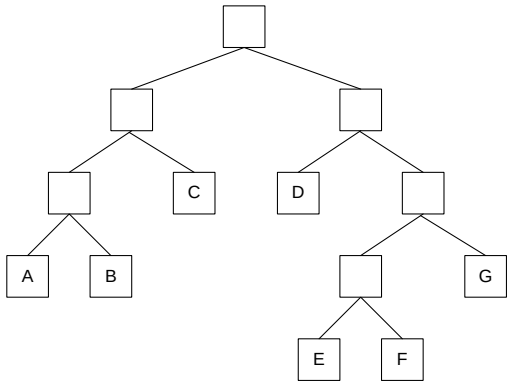
presentation problem 3.4

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 & no repeated sub-trees).

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.

problem 3.5b

Extend problem 3.5, tree-path so it can work on generalised trees (ie: not only binary trees).

problem 3.5c

Extend 2.5b so it takes a predicate (rather than a value) and searches for an item in the tree to satisfy the predicate.

problem 3.5d

Extend problem 3.5c so tree-path returns a function which locates and returns the item you found in the tree.

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/false – 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/false

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.