Sample Questions: SICP Section 2.2

Hierarchical Data and the Closure Property

1. How many cons cells make up the structure with the print value of `(1 (2 3) 4)`?
   (A) 3
   (B) 4
   (C) 5
   (D) 6
2. How many cons cells make up the structure with the print value of `(1 (2 3) . 4)`?
   (A) 3
   (B) 6
   (C) 4
   (D) 5
3. How many cons cells make up the structure with the print value of `(1 (2 . 3) . 4)`?
   (A) 6
   (B) 5
   (C) 4
   (D) 3
4. How many cons cells make up the structure with the print value of `(1 (2 . 3) 4)`?
   (A) 3
   (B) 4
   (C) 5
   (D) 6
5. What is the minimum number of cons cells needed to bundle up 4 numbers into a single cons structure?
   (A) 5
   (B) 6
   (C) 4
   (D) 3
6. What is the minimum number of cons cells needed to bundle up 4 numbers into a single list structure (i.e. no nested lists)?
   (A) 3
   (B) 6
   (C) 5
   (D) 4
7. Draw a cons structure in the style of the text that represents a binary search tree with the elements 3, 1, 2, 5 inserted in that order. Use nils to indicate missing children.
8. Draw a cons structure in the style of the text that represents a trinary search tree with the elements 3, 1, 2, 1, 5, 1 inserted in that order. A trinary search tree behaves just as a binary search tree except duplicates are inserted as middle children. Use nils to indicate missing children.
9. Consider the singly-linked list $2 \rightarrow 5 \rightarrow 3 \rightarrow nil$. Draw a cons cell representation of this list, assuming there is only a head pointer. Use the style of the text. Use the variable *items* to point to this list.
10. Consider the singly-linked list $2 \rightarrow 5 \rightarrow 3 \rightarrow nil$. Draw a cons cell representation of this list, assuming there is a head and a tail pointer. Use the style of the text. Use the variable *items* to point to this list.

11. (3 points) Define a function named *flatten* that takes deeply nested list and returns a flat list of all the elements in the original list. For example, `(flatten '((a (b)) c))` evaluates to `(a b c)`. You may find the functions *pair*? or *atom*? useful.

12. ount the number of.Sum the odd values in a deeply nested list. Pick from the components *flatten*, *map*, *keep*, *remove*, *accumulate*, *append*, and *odd*?. Assume the name of the incoming list is *items*.

13. Suppose I wish to find the product of all the prime numbers from 1 to n? Pick from the components *enumerate*, *map*, *keep*, *remove*, *accumulate*, and *prime*?. Start with n.

14. Suppose I wish to collect all the non-prime numbers from 1 to n? Pick from the components *enumerate*, *map*, *keep*, *accumulate*, and *prime*?. Start with n. Note: *remove* is not available.

15. Sum of all the even numbered squares of the numbers in a given list named *items*. Pick from the components *enumerate*, *map*, *remove*, *accumulate*, *square*, and *even*?. Note: *keep* is not available.

16. Collect all the odd numbered squares of the numbers in a given list named *items*. Pick from the components *enumerate*, *map*, *remove*, *accumulate*, *square*, and *even*?.

17. Collect the squares of all the Mersenne primes generated from a list of exponents, named *raises*. A Mersenne prime has the form $2^n - 1$. Pick from the components *enumerate*, *map*, *keep*, *remove*, *accumulate*, *prime*?, *square*, and *expt*.

18. (2 points) Collect all the pairs (i, j) such that $0 \leq i < n$ and $0 \leq j < n$. Pick from the components *enumerate*, *map*, *keep*, *remove*, *accumulate*, and *expand*. The *expand* function takes a list, a single item, and a location, and creates a list of lists composed each of the list items and the single item. For example, `(expand '(1 4 2) 0 'back)` evaluates to `((1 0) (4 0) (2 0))`, while `(expand '(1 4 2) 0 'front)` evaluates to `((0 1) (0 4) (0 2))`. Start with n.

19. (2 points) Collect all the pairs (i, j) such that $0 \leq i < n$ and $0 \leq j < i$. Pick from the components *enumerate*, *map*, *keep*, *remove*, *accumulate*, and *expand*. The *expand* function takes a list, a single item, and a location, and creates a list of lists composed each of the list items and the single item. For example, `(expand '(1 4 2) 0 'back)` evaluates to `((1 0) (4 0) (2 0))`, while `(expand '(1 4 2) 0 'front)` evaluates to `((0 1) (0 4) (0 2))`. Start with n.

20. (2 points) Collect all the pairs (i, j) such that $0 \leq i < n$ and $i \leq j < n$. Pick from the components *enumerate*, *map*, *keep*, *remove*, *accumulate*, and *expand*. The *expand* function takes a list, a single item, and a location, and creates a list of lists composed each of the list items and the single item. For example, `(expand '(1 4 2) 0 'back)` evaluates to `((1 0) (4 0) (2 0))`, while `(expand '(1 4 2) 0 'front)` evaluates to `((0 1) (0 4) (0 2))`. Start with n.

21. (3 points) Define a function named *index* that retrieves the i^{th} element of a given list. Do not perform any error checking.

22. (3 points) Define a function named *list?* that determines whether a given cons structures is a list at the top level. Do not check if sub-structures are lists. Do not perform any error checking.

23. (3 points) Define a function named *ntail* that returns the n^{th} tail of a given list. When passed a value of 0, *ntail* should return the given list; when passed a value of 1, *ntail* should return the *cdr* of the list, and so on. Do not perform any error checking.

24. (3 points) Define a function named *list+* that returns the sum of all the elements in a given list made up of numbers. The list does not have any sub-lists. Do not perform any error checking.

25. (3 points) Define a function named *list+* that returns the sum of all the elements in a given list made up of numbers. The list may have sub-lists that need to be considered. Assume the existence of the predicate *list?*. Do not perform any error checking.

26. (3 points) Define a function named *oddCount* that returns the number of elements in a given list that are odd. The list does not have any sub-lists. You must use the predefined predicate function *even?*. Do not perform any error checking.

27. (3 points) Define a function named *evenCount* that returns the number of elements in a given list that are even. The list may have sub-lists that need to be considered. You must use the predefined predicates *even?* and *list?*. Do not perform any error checking.

28. (3 points) Define a function named *consCount* that counts the number of cons cells in a given structure. Assume the existence of the predicate *pair?* which returns true if its argument is a cons cell.

29. (3 points) Define a function named *doubleList* that returns a new list composed of the numbers that are twice that in a given list of numbers (in the same order). Do not use *map*. Do not perform any error checking.