

# CS 538

## Project #1

### Programming in Scheme

Due: Friday, March 28, 2008

(Not accepted after Friday, April 4, 2008)

**Handin Instructions.** Place the Scheme code you write to solve this assignment in `~cs538-1/public/handin/proj1/login` where `login` is your login name. Create files 1a, 1b, 2a, 2b, 2c, 2d, 3, 4a, 4b, 5a, 5b and 5c in your handin directory.

1. (a) The `subsets` function we defined in class used functions `extend` and `distrib` to compute all possible subsets of a list of distinct values. However, the implementation we used does not order the subsets in any particular way.  
Rewrite `subsets`, `extend` and `distrib` so that subsets are generated in order of increasing size. That is, the empty subset must come first, followed by all subsets of size one, then all subsets of size two, etc. Any ordering among subsets of the same size is OK, but a smaller subset must always precede a larger one.  
For example `(subsets '(1 2))` may produce `(() (1) (2) (1 2))` or `(() (2) (1) (1 2))` but not `(() (1 2) (2) (1))`.
- (b) The `subsets` function is sometimes used to enumerate all the possible selections from a list of distinct values. Since the number of subsets possible can be *very* large, a predicate is often used to filter out unwanted subsets. If the predicate is true, the subset is kept; if it is false the subset is discarded. For example, a filter predicate might be used to discard subsets that are too large or too expensive.

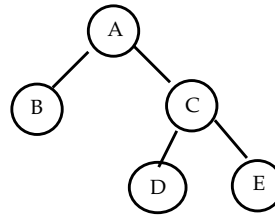
If we use a filter predicate *after* all subsets are generated, we may waste a great deal of time and space generating subsets we don't really want. An alternative is to filter subsets as they are generated. Assume we limit our attention to filter predicates that are *monotone*. This means that if `fp` is false for set `S`, it will also be false for any larger set that contains `S`. Filter functions that limit a set to a given size or cost are monotone.

Write a Scheme function `(filtered-subsets L fp)` that generates all subsets of list `L` for which `fp` is true. This function should filter subsets as they are generated to avoid generation of large numbers of unwanted subsets. Test your solution on

```
(filtered-subsets  
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)  
  (lambda (S) (<= (reduce + S 0) 8)))
```

This call produces all subsets of the first 20 integers that sum to 8 or less (there are 25 such subsets, including the empty subset). This function should execute in just a few seconds, even though there are  $2^{20}$  subsets that might be considered.

2. (a) In Scheme a labeled binary tree can be represented as a list. An empty tree is `()`. A non-empty binary tree, `T`, is represented by a triple `(left lab right)`. `lab` is `T`'s label (this can be any type). `left` is `T`'s left subtree, represented as a list. Similarly, `right` is `T`'s right subtree, again represented as a list. For example, this tree



is represented as `((() B ()) A ((() D ()) C (() E () )))`

Not all lists represent a valid labeled binary tree; only null lists and lists containing a label value between two valid labeled subtree lists are valid.

Write a Scheme function `(valid-bintree? T)` that tests whether tree `T` is valid. For example

```

(valid-bintree? ()) ⇒ #t
(valid-bintree? '(() 1 ()) ) ⇒ #t
(valid-bintree? '(1 () ()) ) ⇒ #f

```

- (b) Two labeled binary trees are equal if their root nodes have the same label and their corresponding left and right subtrees are equal. Write a Scheme function declared as `(equal-bintrees? T1 T2)` that tests whether trees `T1` and `T2` (represented as lists) are equal. If either `T1` or `T2` is invalid, return `#f` (as an error indication). For example

```

(equal-bintrees? () ()) ⇒ #t
(equal-bintrees? '(() 1 ()) '(() 1 ()) ) ⇒ #t
(equal-bintrees? '(() 2 ()) 1 ()) '(() 1 ( () 2 ())) ⇒ #f
(equal-bintrees? '(1 1) '(1 1)) ⇒ #f

```

- (c) There are many ways to visit the nodes of a binary tree. One of the most useful visit orders is **inorder**, which visits the left subtree, then the root, then the right subtree. In the tree shown in part (a), an inorder traversal visits B, then A, then D, C and E.

Write a Scheme function `(tree-map f T)` that visits the nodes of binary tree `T` in inorder. At each node function `f` is applied to the node's value. A list of results is returned. Thus if an inorder traversal of `T` visits nodes `X`, `Y` and `Z`, then a list containing `((f X) (f Y) (f Z))` is returned by `tree-map`. If tree `T` is invalid, return `#f` (as an error indication).

- (d) A labeled binary tree `T` is **ordered** with respect to an ordering relation `le` if
- (i) `T`'s left and right subtrees are themselves ordered.
  - (ii) All labels in `T`'s left subtree are `le` `T`'s label.
  - (iii) `T`'s label is `le` all labels in `T`'s right subtree.
- You are to write a Scheme function `(ordered? T le)` that tests whether the labeled binary tree represented by `T` is ordered using predicate `le` as the ordering relation. If `T` is an invalid labeled binary tree, `#f` should be returned

For example,

```
(ordered?  () <= ) ⇒ #t
(ordered?  '(() 1 ()) 2 ()) <= ) ⇒ #t
(ordered?  '(() 2 (() 3 ())) <= ) ⇒ #t
(ordered?  '(() 1 ()) 2 (() 3 ()) ) <= ) ⇒ #t
(ordered?  '(() 1 ()) 0 (() 3 ()) ) <= ) ⇒ #f
(ordered?  '(1 2 3) <= ) ⇒ #f
```

3. Labeled binary trees may also be represented by Scheme functions rather than a list. An empty tree is represented as #f and a non-empty tree is represented as a function that accepts various tree commands (as described below).

The call (build-tree lab left right) returns a tree function. lab is the tree's label (this can be any type). left and right are the tree's left and right subtrees; these must be tree functions or #f.

If t is a tree function, it must accept the following calls:

(t label)

The label assigned to tree t is returned.

(t left)

The left subtree of tree t is returned. This is either a tree function or #f.

(t right)

The right subtree of tree t is returned. This is either a tree function or #f.

(t equal? u)

Tree t is compared with tree u for equality, using the definition of question 2 (b). u must be a tree function or #f.

(t map f)

Tree t is traversed in inorder. Function f is applied to the label of each node visited, and a list of result values is returned. (This is exactly the same operation as that defined in question 2 (c)).

(t ordered? le)

Tree t is examined to see if it is ordered, using the ordering relation le. (This is exactly the same test as that defined in question 2 (d)).

The following calls illustrate how tree functions are expected to behave:

```
(define (leaf lab) (build-tree lab #f #f))
(define my-tree (build-tree 2 (leaf 1) (leaf 3)))
(my-tree 'label) ⇒ 2
((my-tree 'left) 'label) ⇒ 1
((my-tree 'right) 'label) ⇒ 3
(my-tree 'equal? my-tree) ⇒ #t
(my-tree 'equal? (my-tree 'left)) ⇒ #f
(my-tree 'map (lambda (x) x)) ⇒ (1 2 3)
(my-tree 'ordered? <=) ⇒ #t
(my-tree 'ordered? >=) ⇒ #f
```

4. (a) Write a pair of Scheme functions, `(gen-list start stop step)` and `(pair-sum? list val)`. The function `gen-list` will generate a list of integers, from `start` to `stop`, with consecutive values incremented by `step`. (If `start > stop` then the empty list is generated). For example `(gen-list 1 9 2) ⇒ (1 3 5 7 9)`.

The predicate `pair-sum?` tests whether any two adjacent values in `list` have a sum equal to `val`. For example, `(pair-sum? '(1 2 3) 5) ⇒ #t` since  $2+3=5$ . Similarly, `(pair-sum? (gen-list 1 100 1) 10) ⇒ #f` since no two adjacent integers in the range 1 to 100 have a sum of 10.

- (b) A problem with a function like `pair-sum?` is the fact that its `list` parameter must be fully computed in advance, even if all the values on the list are not needed. For example, `(pair-sum? (gen-list 1 100000 1) 3)` will spend a lot of time and space building up a list of 100000 values, even though only the first two are needed!

An alternative to completely building a complex data structure is **lazy evaluation**. That is, part of the structure is built along with a **suspension**—a function that will supply a few more values upon request. The following two Scheme functions produce a sequence of integers in a lazy manner.

```
(define (return-one-val start stop step)
  (if (> (+ start step) stop)
      (cons start #f)
      (cons start
              (lambda () (return-one-val (+ start step) stop step)))
  )
)

(define (int-seq start stop step)
  (if (> start stop)
      (cons #f #f)
      (return-one-val start stop step)
  )
)
```

When called, `int-seq` returns a pair of values. The *car* is the first integer in the sequence, or `#f` if the sequence is empty. The *cdr* is a function that, when called, will return another pair. That pair consists of the next integer in the sequence plus another suspension function. When the end of the sequence is reached, the *cdr* of the pair is `#f` (rather than a function), indicating that no more values can be produced.

Create a new version of `pair-sum?` called `pair-sum-seq?` that takes a lazy integer sequence (as defined by `int-seq`) rather than a list as a parameter. You can use the Scheme function `(time (f args))` to time the evaluation of `(f args)`. Compare the execution times of `(pair-sum? (gen-list 1 100000 1) 101)` and `(pair-sum-seq? (int-seq 1 100000 1) 101)`. Which is faster? Why? Now compare the execution times of `(pair-sum? (gen-list 1 100000 1) 100)` and `(pair-sum-seq? (int-seq 1 100000 1) 100)`. Again, which is faster, and why?

5. (a) Assume we have a list  $L$  of integers. Define a function `(make-sublists L)` that places each integer in  $L$  into its own sublist (of size one). That is, if  $L$  has  $n$  integers in it, `make-sublists` produces a list of  $n$  sublists, each containing a single integer from  $L$ . For example,

```
(make-sublists ()) ⇒ ()  
(make-sublists '(1 2 3 4)) ⇒ ((1) (2) (3) (4))
```

- (b) Each of the sublists produced by `make-sublists` is trivially sorted. If we merge the first two sublists together into sorted order, we have a sorted sublist of size 2. If we then merge the third and fourth sublists, then the fifth and sixth sublists, etc., we end up with  $n/2$  sorted sublists of size 2, rather than  $n$  sublists of size 1 (the final sublist may be of size 1 if it has no partner to pair with). If we iterate this merging process, we next get  $n/4$  sorted sublists of size 4, then  $n/8$  sorted sublists of size 8, etc. Finally, we produce a single sorted sublist of size  $n$ . This sorting logic is the basis of a **merge sort**. Write a Scheme function `(merge-sort L)` that first divides  $L$  into unit sublists using `make-sublists`, and then repeatedly merges adjacent sublists until a single sorted list is produced. You may create and use any additional functions you find useful or necessary.
- (c) Once a list is sorted, it is easy to test for duplicates—we just compare adjacent values. However, testing for duplicates after the sort is finished is somewhat inefficient. If a duplicate appears in  $L$  it can readily be detected when values from sublists are merged.

Create a function `(merge-sort-cc L)` (and whatever auxiliary functions you require) that makes duplicate checking *integral* to the merge component of the sort. As soon as a duplicate is seen, `#f` should be immediately returned (as an error indication), without any further processing of  $L$ . If no duplicates are found, the sorted values of  $L$  are returned. You should use `call-with-current-continuation` to implement the exception mechanism you will need to return when you see a duplicate value.