Programming Languages (COP 4020/6021)

Assignment I

Objectives

1. To become acquainted with the SML/NJ compiler.
2. To understand basic ML constructs such as lists, functions, pattern matching, anonymous variables, and let-environments.
3. To gain experience defining recursive functions in a functional programming language.

Due Date: Monday, January 22, 2018 (at 5pm)

Machine Details

Complete this assignment by yourself on the following CSEE network computers: c4lab01, c4lab02, ..., c4lab20. These machines are physically located in ENB 220. You can connect to the C4 machines from home using SSH. (Example: Host name: *c4lab01.csee.usf.edu* Login ID and Password: <your NetID username and password>) You are responsible for ensuring that your programs compile and execute properly on these machines.

Assignment Description

0) Read Sections 1-3.6.3 and 4.1-4.2 of the *Elements of ML Programming* textbook.

1) Let's represent finite sets of integers as pairs in ML. The first part of the pair is an `int` indicating the set's size; the second part of the pair is an `int list` containing all the set elements. Hence, our sets have type `int * int list`. For example, the value `(5, [1,3,5,7,9])` represents the set {1, 3, 5, 7, 9}. The order of integers in the `int list` is irrelevant, so `(5, [3,7,1,9,5])` also represents the set {1, 3, 5, 7, 9}.

   For this assignment, two data-structure invariants must be maintained. First, the size of a list must be correct; for example, `(4, [1,3,5,7,9])` and `(6, [1,3,5,7,9])` are invalid sets. Second, no element can appear more than once in the `int list`; for example, `(5, [3,7,1,9,5,3])` and `(6, [3,7,1,9,5,3])` are invalid sets.

   Using this representation of sets, implement the following functions in a file named *as1.sml*:

   (a) `size`
   This function returns the size of a finite set of integers. For example, `size((3,[2,7,4]))` returns `3`.

   (b) `elementof`
   This function tests whether an integer is an element of a given set of integers. For example, `elementof(4, (3,[2,7,4]))` returns `true` because 4 is an element of {2, 7, 4}, while `elementof(3, (3,[2,7,4]))` returns `false`.

   (c) `subset`
   This function tests whether one set is a subset of another. For example, `subset((3,[2,7,4]),(4,[7,9,4,2]))` returns `true` because {2, 7, 4} is a subset of {7, 9, 4, 2}. Note that the empty set `(0,[])` is a subset of every set.

(d) `same`
This function tests whether two sets are equal. For example, `same((3,[2,7,4]), (3,[4,2,7]))` returns `true` because {2, 7, 4} is treated the same as {4, 2, 7}.

(e) `union`
This function returns the union of two sets. For example, `union((3,[2,7,4]), (2,[7,3]))` may return `(4,[4,2,3,7])` because {2, 7, 4} ∪ {7, 3} = {4, 2, 3, 7}.

(f) `lunion`
This function returns the union of every set in a list of sets. When passed an empty list, `lunion` returns the empty set. For example, `lunion([(3,[2,7,4]), (2,[7,3]), (0,[])])` may return `(4,[4,2,3,7])` because {2, 7, 4} ∪ {7, 3} ∪ ∅ = {4, 2, 3, 7}.

(g) `intersection`
This function returns the intersection of two sets. For example, `intersection( (3,[2,7,4]), (2,[7,3]))` returns `(1,[7])` because {2, 7, 4} ∩ {7, 3} = {7}.

(h) `lintersection`
This function returns the intersection of every set in a list of sets. When passed an empty list, `lintersection` returns the empty set. For example, `lintersection( [(3,[2,7,4]), (2,[7,3]), (0,[])])` returns `(0,[])` because {2, 7, 4} ∩ {7, 3} ∩ ∅ = ∅.

(i) `powerset`
This function takes a set of integers *S* and returns a list of the subsets of *S*. For example, `powerset((3,[2,7,4]))` may return `[(0,[]),(1,[2]),(1,[7]),(1,[4]), (2,[2,7]),(2,[2,4]),(2,[7,4]),(3,[2,7,4])]` because $2^{\{2,7,4\}}$ = {∅, {2}, {7}, {4}, {2,7}, {2,4}, {7,4}, {2,7,4}}. The order of elements in lists returned by `powerset` is irrelevant.

(j) `ipowerset`
This function takes a list *L* of sets and tests whether there exists a set *S* such that `powerset`(*S*)=*L*. If such an *S* exists then `ipowerset` returns `SOME` *S*; otherwise `ipowerset` returns `NONE`. For example, `ipowerset([(1,[3]),(1,[7]), (0,[]),(2,[7,3])])` may return `SOME (2,[7,3])` because $2^{\{7,3\}}$ = {{3}, {7}, ∅, {7,3}}, while `ipowerset([(1,[3]),(1,[7]),(2,[7,3])])` returns `NONE`.

(k) `product` [Note: For undergraduates, this function is extra credit, worth up to +10%.]
This function takes a list of sets S_1, S_2, ..., S_n and returns a list of all the lists that contain an element of S_1 in their first position, an element of S_2 in their second position, etc. If one of $S_1..S_n$ is the empty set, then `product` simply returns `[]`, and if `product` is called on an empty-list argument, then it returns `[[]]`. For example, `product([ (3,[2,7,4]),(2,[7,3])])` may return `[[2,7],[2,3],[7,7],[7,3], [4,7],[4,3]]` because {2, 7, 4} x {7, 3} = {(2,7), (2,3), (7,7), (7,3), (4,7), (4,3)}. The order of lists returned by `product` is irrelevant, so `product([(3,[2,7,4]), (2,[7,3])])` could alternatively return `[[4,3],[4,7],[7,3],[7,7],[2,3], [2,7]]`, etc.

Sample Execution (please remember that the order of elements in sets is irrelevant)

```
> sml
Standard ML of New Jersey v110.74 [built: Thu Aug 16 11:25:45 2012]
- use "as1.sml";
[opening as1.sml]
val size = fn : int * int list -> int
val elementof = fn : int * (int * int list) -> bool
val subset = fn : (int * int list) * (int * int list) -> bool
val same = fn : (int * int list) * (int * int list) -> bool
val union = fn : (int * int list) * (int * int list) -> int * int list
val lunion = fn : (int * int list) list -> int * int list
val intersection = fn : (int * int list) * (int * int list) -> int * int list
val lintersection = fn : (int * int list) list -> int * int list
val powerset = fn : int * int list -> (int * int list) list
val ipowerset = fn : (int * int list) list -> (int * int list) option
val product = fn : (int * int list) list -> int list list
val it = () : unit
- val E = (0,[]);
val E = (0,[]) : int * 'a list
- val S = (3,[2,7,4]);
val S = (3,[2,7,4]) : int * int list
- val T = (4,[7,9,4,2]);
val T = (4,[7,9,4,2]) : int * int list
- val S' = (3,[4,2,7]);
val S' = (3,[4,2,7]) : int * int list
- val U = (2,[7,3]);
val U = (2,[7,3]) : int * int list
- size(S);
val it = 3 : int
- elementof(4,S);
val it = true : bool
- elementof(0,E);
val it = false : bool
- subset(S,T);
val it = true : bool
- subset(T,S);
val it = false : bool
- same(S,S');
val it = true : bool
- same(S,E);
val it = false : bool
- union(S,U);
val it = (4,[4,2,7,3]) : int * int list
- lunion([S,T,S',U,E]);
val it = (5,[9,2,4,3,7]) : int * int list
- lunion([]);
val it = (0,[]) : int * int list
- intersection(S,U);
val it = (1,[7]) : int * int list
- intersection(S,E);
val it = (0,[]) : int * int list
- lintersection([S,U]);
val it = (1,[7]) : int * int list
- lintersection([]);
val it = (0,[]) : int * int list
- powerset(E);
val it = [(0,[])] : (int * int list) list
- val P = powerset(S);
val P =
  [(0,[]),(1,[2]),(1,[7]),(2,[2,7]),(1,[4]),(2,[2,4]),(2,[7,4]),(3,[2,7,4])]
  : (int * int list) list
- ipowerset(P);
val it = SOME (3,[2,7,4]) : (int * int list) option
```

```
- ipowerset([E]);
val it = SOME (0,[]) : (int * int list) option
- ipowerset([(0,[]),(1,[2]),(1,[7])]);
val it = NONE : (int * int list) option
- product([S,U]);
val it = [[2,7],[2,3],[7,7],[7,3],[4,7],[4,3]] : int list list
- product([]);
val it = [[]] : int list list
- product([S,U,E,T]);
val it = [] : int list list
```

Grading

For full credit, your implementation must:

- be commented and formatted appropriately. To make it easier for the TA to grade, use spaces instead of tabs for indentation.
- use anonymous variables, pattern matching, and let-environments when appropriate (e.g., define all helper functions in let-environments).
- compile on the C4 machines with no errors or warnings.
- not use any ML features that cause *side effects* to occur (e.g., I/O or pointer use).
- not use any built-in (i.e., predefined or library) functions.
- not be significantly more complicated than is necessary.
- assume that incoming set arguments satisfy the data-structure invariants described on Page 1 (i.e., your functions should not try to enforce the validity of their set arguments).
- never pass as an argument a set that violates the data-structure invariants described on Page 1 (i.e., when invoking a function, your code may never pass an invalid set as an argument).
- never return from a function a set that violates the data-structure invariants described on Page 1.

Please note that we will test submissions on inputs not shown in the sample execution above.

Hints

It took me 1-2 hours to implement and test my *as1.sml*, which is 46 lines of code (not counting whitespace/comments). If, after completely reading Sections 1-3.6.3 and 4.1-4.2 of the textbook, you find yourself spending a significant amount of time (e.g., more than 12 hours) on this assignment, please visit or email the teaching assistant to ask for help with whatever problems you are having.

Submission Notes

- Type the following pledge as an initial comment in your *as1.sml* file: "I pledge my Honor that I have not cheated, and will not cheat, on this assignment." Type your name after the pledge. Not including this pledge will lower your grade 50%.
- Upload and submit your *as1.sml* file into the Canvas folder for this assignment.
- You may submit your assignment in Canvas as many times as you like; we will grade your latest submission.
- You may submit your assignment late (between 5pm on 1/22 and 5pm on 1/24) with a 15% penalty.