Edward Fleri Soler

CSCI 401 Programming Assignment 1

Notice: All code and documentation within this paper is the work of Edward Fleri Soler, and none of this work has been extracted from external sources.

- **Required Utilities**

The following lines of code define tools which would later be used to define the remaining problems:

- member?

```
#lang racket

(define member?
  (lambda (x list)
    (cond ((null? list) #f)
          ((equal? x (car list)) #t)
          (else (member? x (cdr list))))))
```

The 'member?' definition takes two arguments; an element and a set. This definition returns true if the provided element is a member of the provided set. This definition runs through the provided set, trying to match the provided element with the element at the head of the set. If the head of the set matches the element, true is returned, otherwise the definition is recursively called to work upon the tail (remainder) of the set. If the end of the set is reached (null), the definition returns false, as it exhaustively searched through the set without finding a matching element.

- Subset?

```
(define Subset?
  (lambda (l1 l2)
    (cond ((null? l1) #t)
          ((member? (car l1) l2) (Subset? (cdr l1) l2))
          (else #f))))
```

The 'Subset?' definition takes two sets as arguments and tests whether the first set 'l1' is a subset of the second set 'l2'. In a recursive fashion, this program runs through each element in the first set, testing whether it is a member of the second set via the previously programmed 'member?' definition. If this is the case, the program recursively calls itself, passing the tail of the first set as an argument together with the second set. This is repeated until an element in the first set is not a part of the second, in which case 'false' is returned. Otherwise, if the end of the first set is reached (null), then 'true' is returned, as each element is the first set has been found to be a member in the second.

- remDup

```
(define remDup
  (lambda (l)
    (cond ((null? l) '())
          ((member? (car l) (cdr l)) (remDup (cdr l)))
          (else (cons (car l) (remDup (cdr l)))))))
```

This program is designed to remove duplicate elements from a list, making it a set. This will later be used for set union. This definition takes as a list as an argument. The first element in each list is tested for membership in the rest of the list. If this results as true, it means that duplicate elements exist in the set. In this case, the method is recursively called and passed the tail of the list, excluding the head and thus removing it. Else, if the element at the head of the list is not repeated throughout the tail of the list, it is kept to be reconnected to the rest of the list via the 'cons' instruction. Once the program descends the recursive levels, the unique elements in the list will be added back to the remainder of the list.

- union

```
(define union
  (lambda (l1 l2)
    (remDup (append l1 l2))))
```

This program performs set union on two provided sets. Two sets 'l1' and 'l2' are passed as arguments. These sets are appended to each other via the inbuilt 'append' function. However, the newly created set may contain duplicates, so the 'remDup' function is used to remove any duplicates.

- back

```
(define back
    (lambda (x list)
      (cond ((null? list) '())
            ((equal? (car (cdr(car list))) x) (cons (car (car
list)) (back x (cdr list))))
            (else (back x (cdr list))))))
```

'back' is a definition which is required for the other definitions at the end of this paper. This program accepts as arguments an element of a set and a set. It is designed to work upon sets of binary pairs. The purpose behind this program is to return a list of all first element in the binary pairs with the second element being 'x'. Therefore, if the following line is executed:

(back '1 '((2 1) (1 3) (4 1) (4 4)))

The result would be the set (2 4).

This program works by taking the pair at the head of the set and comparing the second element in the pair to x. If the second element in the pair is equal to x, then the first element in the pair is to be included in the list to be returned. This, once again, is done via the 'cons' function, which appends the element to the front of the list, with the rest of the list being the return value of the recursive call. If the second element in the pair is not equal to x, then the first element in the pair is discarded and the program recursively calls itself, passing x and the tail of the list as arguments. Once the end of the list is reached, the recursion ends, as the whole list has been inspected.

- front

```
(define front
    (lambda (x list)
      (cond ((null? list) '())
            ((equal? (car (car list)) x) (cons (car (cdr (car
list))) (front x (cdr list))))
            (else (front x (cdr list))))))
```

As is the case with the 'back' program, the 'front' program is required to define future problems which will be seen in the rest of the paper. This program also accepts an element of a set and a set as its arguments, and performs the exact same task as the 'back' script does, with the exception that it compares the provided element to the first element within the binary pairs, returning the second element within the binary pair when a match is found.

In exactly the same manner, the pair at the front of the set is focused upon, and its leading element within the binary pair is compared to the given element 'x'. In the case of a match, the second element within the pair is output together with the other corresponding elements. A recursive call is then made, passing the tail of the list as the set parameter. This is repeated until the whole set is explored.

- product

```
(define product
  (lambda (x l)
         (cond ((or (null? x) (null? l)) '())
               (else (append (list(list x (car l))) (product x
(cdr l)))))))
```

This utility, like those above, serve a future purpose in defining problems. This program takes as inputs an element 'x' and a list 'l' and returns a set of binary pairs, with the first element in the binary pair being the given 'x' and the second element being an element of the set. Through the append and list utilities, the element x is paired with the head of the list 'l' to form a binary pair. This is in turn appended to a list, so as to create a set of sets. Tail recursion then passes on the list, minus the head, repeating

the cycle until the null set is passed. The scope of this utility would be better understood once ‘product 2’ is tackled.

- product2

```
(define product2
  (lambda (l1 l2)
         (cond ((or (null? l1) (null? l2)) '())
               (else (append (product (car l1) l2) (product2
(cdr l1) l2))))))
```

The purpose of this definition is to create a utility called product 2, which performs the cartesian product of two given sets. As inputs, this utility takes two sets, ‘l1’ and ‘l2’. By invoking the previously defined utility ‘product’ and passing it the head of the first list together with the whole of the second list, the cartesian product of the two lists may slowly be formed. This is performed once for each element in the set ‘l1’. The recursive call is passed the rest of the set, save the element at the head. The result returned from ‘product’ is then used to generate the final list, which is output once the stopping condition (‘l1’ being null) is reached. This utility will play a vital role in problems to come.

1. Transitive?

```
(define tempTransitive?
  (lambda (list1 list2)
    (cond ((null? list1) #t)
          ((not (Subset? (product2 (back (car (cdr (car list1)))
list2) (front (car (cdr (car list1))) list2)) list2)) #f)
          (else (tempTransitive? (cdr list1) list2)))))

(define Transitive?
  (lambda (list)
    (tempTransitive? list list)))
```

The purpose of this function is to determine whether a given set is transitive or not. The program ‘Transitive?’ is simply used to invoke the helper function ‘tempTransitive?’, passing it the same set twice as arguments.

The method ‘tempTransitive?’ is defined to accept two sets of sets as parameters. Firstly, the first set named ‘list1’ is tested for being null. This is the stopping condition in the recursive call.

Next, the second element in the binary pair, in the head of the set is extracted and passed as a parameter to the ‘back’ utility, together with the second parameter ‘list2’. This will return a set of all elements in ‘list2’ which are at the front of the binary pair with the given value. Therefore if the given element is 2 and the given set is ((1 2) (2 3) (7 2), this function will return the set (1 7).

Similarly, the ‘front’ utility is passed the same value as is passed to ‘back’ together with the same list. This, similarly, will return a set of elements who are at the back of the binary pair with the given element. Therefore if the given element is 3 and the set is ((3 99) (2 3) (1 4)), this utility will return (99).

The returned sets from these two functions will then be passed as parameters to the ‘product2’ utility, which will return the binary product of the two sets. Therefore if the given parameters are (1 3) and (8 2), the returned set will consist of the following:

((1 8) (1 2) (3 8) (3 2))

This set is then tested to be a subset of the set ‘list2’ via the ‘Subset?’ function. If this set is not a subset of the ‘list2’ set, then it may be concluded that the given set is not transitive, as all required binary pairs for the set to be considered transitive are not within the original set. Therefore false will be returned.

If the subset utility returns true, then the first test for transitivty has been passed and we make a recursive call so as to further test the set. The parameters for the recursive call are the sets ‘list1’ with the first element removed, aswell as the original set ‘list2’, left untouched. Therefore, one may understand that the purpose of having two sets as parameters is so as to leave the second ‘original’ set untouched, while reducing the first set ‘list1’ so as to keep track of which elements have been tested, and which haven’t.

This cycle goes on until a subsequent subset test fails, concluding that the set is not transitive, or until the first set ‘list1’ is exhaustively searched and therefore set to null. In this case, the set is confirmed to be transitive.

2. Anti-Transitive?

```
(define tempAnti-Transitive?
  (lambda (list1 list2)
    (cond ((or (null? list1) (null? list2)) #t)
          ((and (Subset? (product2 (back (car (cdr (car list1))))
list2) (front (car (cdr (car list1))) list2)) list2) (not (null?
(product2 (back (car (cdr (car list1))) list2) (front (car (cdr
(car list1))) list2))))) #f)
          (else (tempAnti-Transitive? (cdr list1) list2)))))

(define Anti-Transitive?
  (lambda (list)
    (tempAnti-Transitive? list list)))
```

The purpose of this function is to test for anti-transitivity; meaning that absolutely no transitivities exist within a given set. This is not as simple as testing for a set to be non-transitive, as that would mean that a set is neither transitive nor anti-transitive.

However, each element within the given set must be tested in a similar fashion to the 'Transitive?' case.

The 'Anti-Transitive?' function simply invokes the 'tempAnti-Transitive?' function, passing it the same given set twice as its parameters.

The helper functon begins by testing whether the given parameters are null, in which case the function can immediately return true. This line of code also serves as the stopping point of the recursion.

Next, the function goes on to perform the exact same procedure as the 'tempTransitive?' function, using the utilities 'product2', 'front' and 'back to generate a set of binary pairs, relating to the pairs required to be in the original set for transitivity to be true. If this set is found to be a subset of the given set, and this set is not null, then the given set is definitely not anti-transitive, as it contains atleast one transitivity. It is, however, important, to test that the generated set is not null, as otherwise if it is null (meaning that no possible transitivities can exist, involving the given element) then it is trivially a subset of any set, passing this condition and thus failing the anti-transitivity test on false grounds.

If control passes this stage of the program, a recursive call is made, passing the first set 'list1', save the head of the set, and the second set 'list2' as parameters. This cycle continues until the set is found to be non anti-transitive, or until the first set 'list1' is null, in which case the test is passed.

For the exact same reason discussed in 'Transitive?' it is vital to make use of two parameters when calling the 'tempAnti-Transitive?' function recursively, so as to keep track of the elements tested.

3. Irreflexive?

```
(define Irreflexive?
  (lambda (list)
    (cond ((null? list) #t)
          ((equal? (car (car list)) (car (cdr (car list)))) #f)
          (else (Irreflexive? (cdr list))))))
```

This function is designed to test whether a given set is irreflexive or not, and does not require a helper function. Taking a set of binary pairs as a parameter, this function goes on to examine the first and second element within each binary pair. If the first element in a binary pair is the same (equal?) as the second, then the set is found to be non-irreflexive, and a return value of false is computed.

In the case of the two elements being different, this cycle is repeated recursively, passing the given set, save the head, as a parameter. If the given argument is found to be null, the given set has been exhausted showing that the set is in fact irreflexive.

4. Composure

```
(define tempCompose
  (lambda (t1 l1 l2)
    (cond ((null? t1) '())
          (else (append  (product2 (back (car (cdr (car t1))) l1) (front (car (cdr (car t1))) l2)) (tempCompose (cdr t1) l1 l2))))))

(define Compose
  (lambda (l1 l2)
    (union (tempCompose l1 l1 l2) (tempCompose l2 l1 l2))))
```

The function ‘Compose’ accepts two sets of binary pairs as arguments, and in-turn invokes the helper function ‘tempCompose’. ‘Compose’ returns the composure of the two given sets.

The helper function ‘tempCompose’ accepts three sets of bianry pairs as arguments. This function goes on to use the utilities ‘back’, ‘front’ and ‘product2’ in a similar fashion to the previously discussed functions, so as to procude a cartesian product of two sets; each set being formed by selecting elements which are paired up with a given element within a binary pair.

A recursive call is then made, passing the tail of the first set together with the other two sets as arguments. This continues until the first argument is found to be null (stopping point). The returned values of the recursive call are then appended to the results of the previous recursive level, ultimately producing the composure of the two given sets.

An extra parameter is required within the helper function, as the last two sets are the original ‘untouched’ sets of which we are finding the compose, while the first set is used to keep track of which elements have been processed.