CSC 351 - Comparative Programming Languages

Project 2 - Scheme

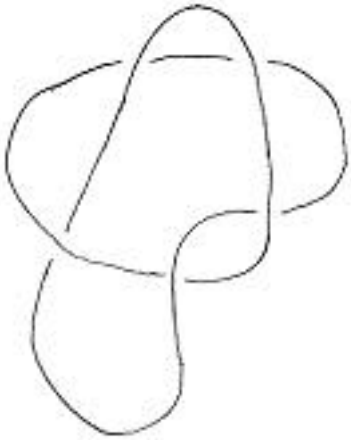
Due: November 14, 2000 11:30 am

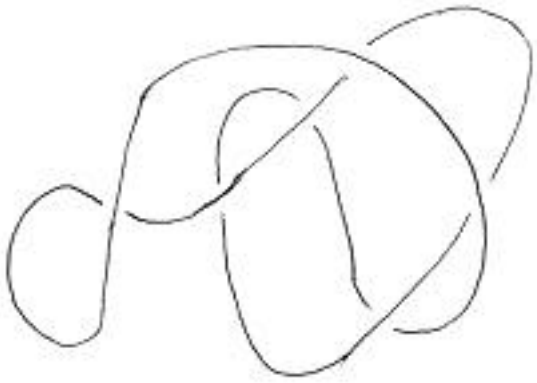
Purpose

This project will give you additional experience with functional programming in Scheme through the exploration of untangling knots.

Background

Imagine that you have a piece of rope that has been tangled and then the two ends have been connected together. The question that can be asked is if the result is a tangle or a knot. If it is a tangle, all of the twists can be undone so that there is a single loop of rope. If it is a knot, there will be no way to remove all of the twists without cutting the rope. In the following two examples, you should see that one is just a tangle and that the other is a knot that cannot be unraveled. Can you tell which is which? In these diagrams, the broken line indicates points where the rope passes under itself.

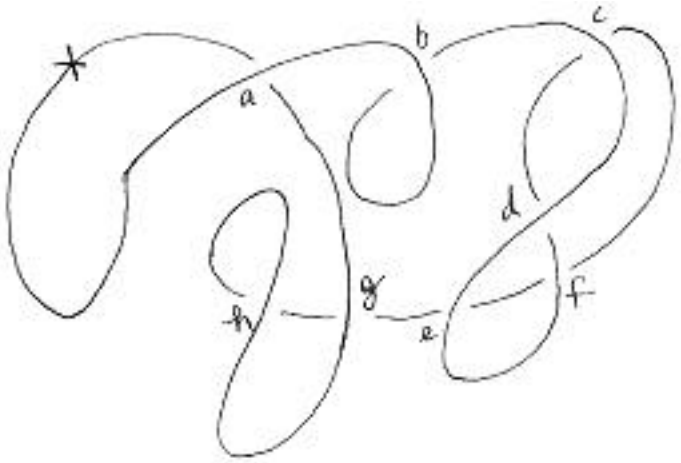




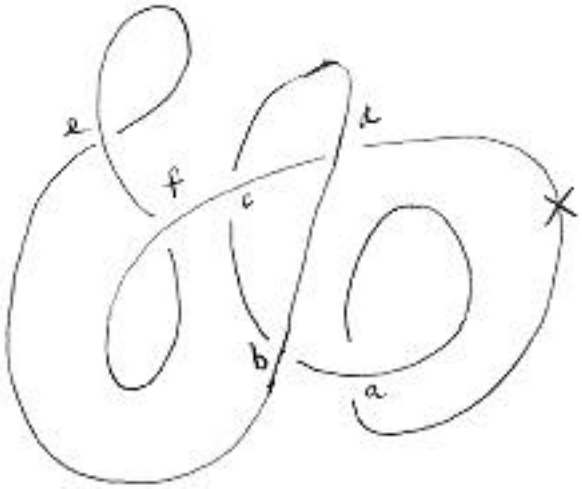
Karl Reidemeister showed in 1948 that any tangle can be undone by performing a series of only three types of moves, now called Reidemeister Moves.

Type I move Type II move Type III move

The first two examples above are called knot diagrams. For our purposes, we will label these diagrams with a set of letters. We first identify a starting point and direction and then move around the rope, labeling the crossings with successive letters of the alphabet. Each crossing gets labeled once. We now can give a list description or a tripcode of each "knot" by again traversing the rope and writing down the labels followed by an 'o' if the rope goes over at that point, and a 'u' of the rope goes under. Each "knot" described in this way will have an even number of pairs since the rope passes each crossing twice, once going over and once going under. You should note that this means a properly formed tripcode will have exactly two pairs for each label and one of those pairs will have an 'o' and the other will have a 'u' as its second element. The following are two more examples showing the corresponding tripcodes.



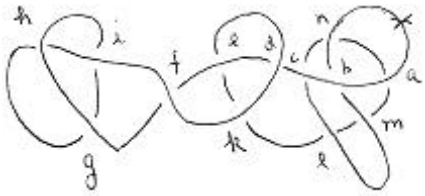
((a o) (b o) (b u) (c o) (d o) (e o) (f o) (d u) (c u) (f u) (e u) (g u) (h u) (h o) (g o) (a u))



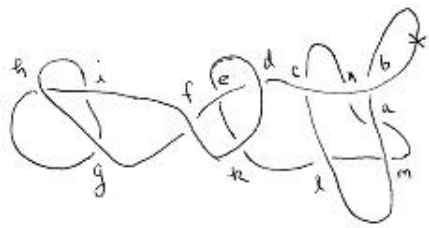
((a u) (a o) (b u) (c u) (d o) (b o) (e u) (e o) (f u) (f o) (c o) (d u))

To show how the Reidemeister Moves work, we will use them to attempt to untangle an example. The move, and where it is applied, used to get the next stage are given under each diagram. The tripcode of each diagram is also given.

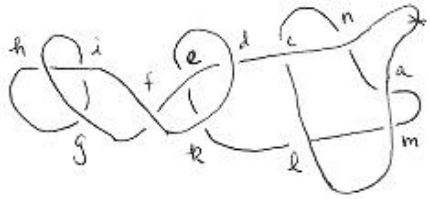
((a o)(b o)(c o)(d u)(e o)(f u)(g o)
(h o)(i u)(g u)(j u)(j o)(h u)(i o)(f o)
(k o)(d o)(e u)(k u)(l u)(m u)(a u)
(n u)(c u)(l o)(m o)(b u)(n o))
Type I (j)



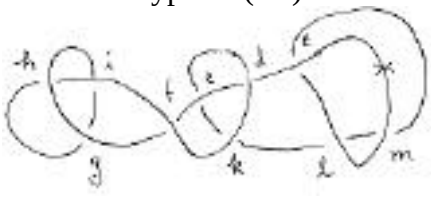
((a o)(b o)(c o)(d u)(e o)(f u)(g o)
(h o)(i u)(g u) (h u)(i o)(f o)
(k o)(d o)(e u)(k u)(l u)(m u)(a u)
(n u)(c u)(l o)(m o)(b u)(n o))
Type III (a/n)



((b o)(n o)(c o)(d u)(e o)(f u)(g o)
(h o)(i u)(g u)(h u)(i o)(f o)(k o)
(d o)(e u)(k u)(l u)(m u)(a u)(n u)
(c u)(l o)(m o)(a o)(b u))
Type I (b)

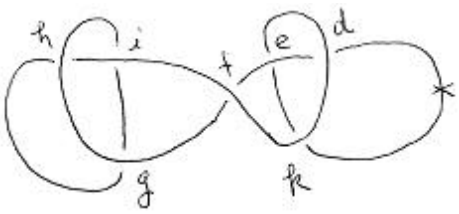


((n o)(c o)(d u)(e o)(f u)(g o)(h o)
(i u)(g u)(h u)(i o)(f o)(k o)(d o)
(e u)(k u)(l u)(m u)(a u)(n u)(c u)
(l o)(m o)(a o))
Type II (a/n)

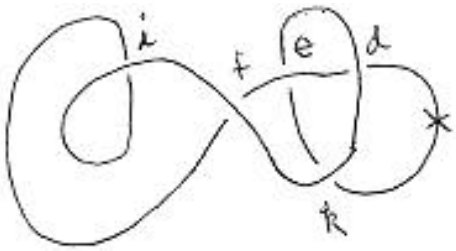


((c o)(d u)(e o)(f u)(g o)(h o)(i u)
(g u)(h u)(i o)(f o)(k o)(d o)(e u)
(k u)(l u)(m u)(c u)(l o)(m o))
Type II (l/m)

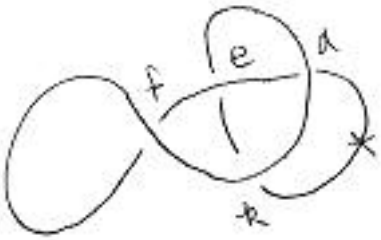
((c o)(d u)(e o)(f u)(g o)(h o)(i u)
(g u)(h u)(i o)(f o)(k o)(d o)(e u)
(k u)(c u))
Type I (c)



((d u)(e o)(f u)(g o)(h o)(i u)(g u)
(h u)(i o)(f o)(k o)(d o)(e u)(k u))
Type II (g/h)

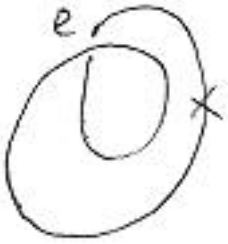


((d u)(e o)(f u)(i u)(i o)(f o)(k o)
(d o)(e u)(k u))
Type I (i)

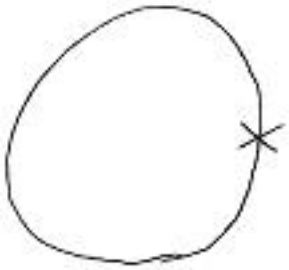


((d u)(e o)(f u)(f o)(k o)(d o)(e u)
(k u))
Type I (f)

((d u)(e o)(k o)(d o)(e u)(k u))
Type II (d/k)



((e o)(e u))
Type I (e)



( )

The careful observer will notice that in the above example, Type I moves are applied at places where there were adjacent pairs of the form (x o) and (x u) in either order. Type II moves are applied at places were there are adjacent pairs of the form (x o) and (y o) followed someplace in the tripcode by the adjacent pairs (x u) and (y u) in some order. (The under crossings could also come before the over crossings. The only requirement is that there is nothing between the two over or two under crossings.) Each Type I and Type II move removes part of the tripcode. If at some point, we can no longer apply any moves, then we must have started with a knot. If all of the pairs are removed, it must have been a tangle. There are two other things that you should notice. One move may create the possibility of additional moves, and the list is really circular, so that the first pair is considered adjacent to the last pair. Type III Moves are not discussed because they do not reduce the tripcode, but just re-arrange it. If you choose to implement Type III Moves, you should have the user decide whether to apply it. Since a Type III Move can undo a previous Type III Move, you can get into an infinite loop without user control.

Detailed Description

You are to write a program in Scheme that will first check that the tripcode is of the correct form, and then try to untangle it using Type I and Type II Moves. (Type III Moves can be implemented for extra credit.) Your program should ask for a filename, and then read the tripcode from that file. It should print out the new tripcode after each move is applied. At the end, you should print out a statement as to whether this is a knot or a tangle.

1) Write the function isAknot that checks if the tripcode is correct. It should check that the list has only pairs, that the number of pairs is even, that each pair has an ‘o’ or ‘u’ as its second element, and that each label only appears in exactly two pairs (one with an ‘o’ and the other with a ‘u’). You will probably want to write subfunctions that test each one of these four requirements. To do the last of these four, you will want to use the map function to get the labels and then produce a list of unique labels. You can then traverse this list of labels and make sure that each appears properly in the tripcode.

2) You will want to modify the removePairs function from lab exercise 5 so that it removes tripcode pairs that represent Type I Moves. This function should also be changed so it removes only one pair at a time, since you have to print the tripcode after each move.

3) You will also need the rotate function of lab 5 as well. You will want to do a single rotation if there are no Type I or Type II moves. If you have no moves, you rotate once and still have no moves, you can stop. Additional rotations will not be of any help.

4) Write two functions that will look for and make a Type I move or a Type II move.

5) Write a main function that will get the tripcode, check if it is valid, and then repeatedly make moves of Type I and II (possibly rotating) until either the tripcode doesn’t change or it becomes the empty list. If you try a Type I Move, try a Type II Move, rotate once, try a Type I Move, try a Type II Move, and the list is not reduced, then this is a knot.

Limitation: All of your code is to change the tripcode as part of the recursive process. You are **not** to use remove or replace functions to change the tripcode.

Deliverables

You are to submit your commented code, a user’s and systems’ manual, and sample input and output. You are also to submit your code and sample input to the instructor electronically by the due date.