An Effective Lisp Project for a Programming Languages Course

Marsha Meredith
Department of Mathematics/Computer Science

Blackburn College
Carlinville IL 62626

Introduction

One focus of many undergraduate courses in Programming Languages is to give students a sense of the character and structure of a variety of languages. To achieve this goal, instructors need programming projects that are small but the solution of which compels students to explore a given language -- to program in it rather than with it. A good project has the following features:

1) It is easily stated. Students should be able to understand the goal of the project without much difficulty; they will face enough confusion with language issues.

2) It is suited to the language. Completion of the project causes students to engage in programming in the style of a particular language.

3) The resulting program is compact. That is, it should be relatively short yet encompass the major elements of the target language. This is not, after all, a software engineering course or an endurance test. Success is important.

4) It should be interesting. This may encourage good students to explore the language further.

5) It should be extensible. The interested student can then use the project as a vehicle for exploring both the language and the application area.

A project with which we have had some success in the last two years derives from ongoing departmental research in knot theory and serves as a vehicle for the exploration of Lisp. Students who have had introductory laboratory work in Lisp can generally complete the project within a week or two.

Knot Foundations

If we take a length of string, twist it around itself in space, and join the two ends, we have formed a tangle. The question is, can that tangle be converted into a simple loop without cutting the string? If it can, we say the tangle is an unknot; otherwise it is a knot. Mathematicians who study knots often work with two-dimensional projections, called knot diagrams. Such a projection is shown below.

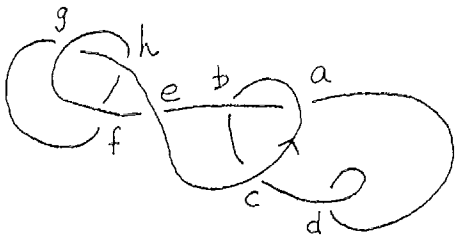


Figure 1 -- A knot diagram.

The letters label crossings, the places where one portion of the string passes over or under another. The "under" string is indicated by a broken line in the diagram. We will use the term arc to denote the piece of string between two crossings.

Karl Reidemeister (1948) showed that any unknot can be untangled by performing an appropriate series of only three types of moves, called

Reidemeister moves. Type-I involves removing a twist from an arc. Type-II involves passing two arcs, one of which lies above the other, across each other. Type-III involves passing an arc that lies above or below the crossing formed by two others past that crossing. Diagrams illustrating the three move types follow.

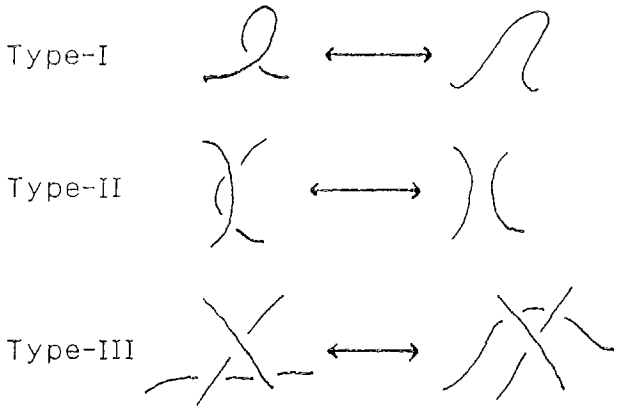


Figure 2 -- Reidemeister moves.

We can show that the tangle of Figure 1 is an unknot by following the series of Reidemeister moves given in Figure 3. Here, only Type-I and Type-II moves are required.

Bringing Knots to Lisp

Randall Weiss (1987) has developed a representation for knot diagrams which he calls a tripcode, an ordered list of pairs <crossing-name, crossing-type> which indicates the order in which crossings are visited during a continuous circuit of the diagram. The crossing-name is a simple identifier such as "a" in the knot diagram in Figure 1; the crossing-type is either "o" (for "over") or "u" (for "under"). Thus, a tripcode for the knot diagram in Figure 1 is: ((a o)(b u)(c u)(d o)(d u)(a u) (b o)(e u)(f o)(g o)(h u)(f u)(g u)(h o) (e o)(c o)).

The presence of a Reidemeister Type-I move in a knot diagram is indicated by the presence in the tripcode of adjacent pairs with identical crossing-names. Removal of the twist in the diagram amounts to removal of those pairs from the tripcode. For example, the first twist removed in Figure 3 was at crossing d. In the tripcode we find the adjacent pairs (d o) (d u) to

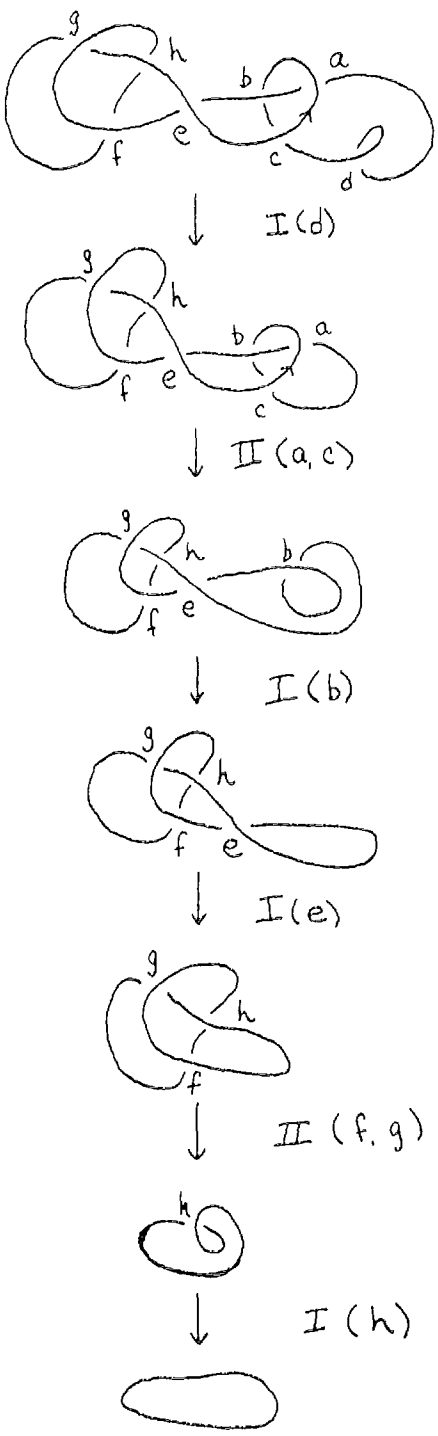


Figure 3 -- Unknotting a tangle.

indicate the same move. We perform the move by deleting these pairs from the tripcode.

Similarly, a Reidemeister Type-II move corresponds to finding two crossing-names with like crossing-types in adjacent pairs twice in the tripcode. For example, in our sample tripcode we find the pairs
(...(f o) (g o)...(f u)(g u)...)
(note that the order of the pairs is immaterial -- had we seen (g u)(f u), there would still be a Type-II move). We perform the Type-II move by removing all four entries from the tripcode.

Project Description

The project we assign our students requires them to represent a knot diagram and perform any available Type-I and Type-II moves. Some extensions of varying difficulty are suggested. The text follows.

Write a Lisp program that accepts a tripcode as input and does the following:

(1) It determines whether or not the tripcode is "syntactically correct". Each crossing-name should occur exactly twice in the tripcode, with opposite crossing-types in the two occurrences.

(2) It finds and performs all Reidemeister Type-I and Type-II moves, and displays the type of move and the crossing(s) involved as the moves are performed.

(3) If the tripcode becomes empty, there are no crossings left. This means your program has found an unknot. It should tell you so.

(4) If the tripcode is non-empty but no further moves are available, the program should print out the remaining tripcode.

Notes: Performance of one move may create another.
The first-listed and last-listed crossings in a tripcode are adjacent.

Extensions: (1) A non-empty tripcode with no Type-I or Type-II moves is a knot if its crossing-types alternate between "o" and "u". Have your program check for this condition and report on the result.

(2) Type-III moves are also detectable from the tripcode. Have the program find and perform Type-III moves under your control. Be aware that Type-III moves (i) do not shorten the tripcode, (ii) lead to their own inverse, and (iii) involve changing the order of tripcode entries.

Conclusions

The project described in this paper meets our requirements for a good Programming Languages project. It would certainly be easier to assign a Towers-of-Hanoi-in-Lisp project because students are already familiar with the problem (and in fact this is a good "Rosetta-stone" project). However, it takes only half an hour to introduce the background necessary to understand the knot project presented here -- time well spent when we consider the criteria of interest and extensibility. Once students have this background the project is, as we have shown, briefly stated.

Certainly, the tripcode structure cries out to be a Lisp list. The operations required to perform the various comparisons, checks, and deletions are vintage Lisp, and include selector and help functions, recursion, search, deletion, and (for Type-III) insertion functions. Moreover, the novelty of the application discourages any tendency to develop the program in Pascal (or some other first language) and translate. Students will use Lisp.

Our students have found the project interesting and not so long and complicated as to be intimidating. Once they have mastered the syntax and functional programming style of Lisp they progress rapidly. The resulting code is only about three pages long. One student became interested and proficient enough to do summer research in knot theory with members of the faculty.

It should be noted that, while Weiss (1990) has recently proposed an unknot detection algorithm based on a generalization of the Reidemeister moves, it is as yet an unsolved problem to determine, in general, when two knots are equivalent. The interested student has open research ground ahead.

References

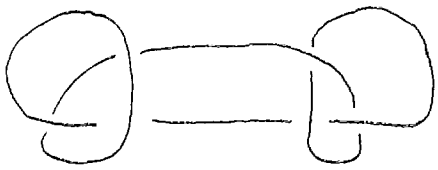
Peterson, I. (1988). The Mathematical Tourist. (New York: W.H. Freeman).

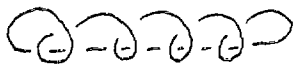
Reidemeister, K. (1948). Knotentheorie. (New York: Chelsea Publishing Co.).

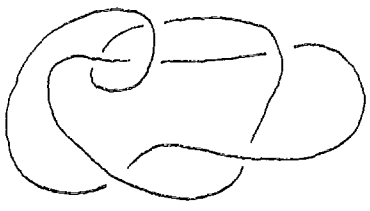
Weiss, R. (1987). Detecting ribbon knots. unpublished dissertation, University of Illinois at Chicago.

Weiss, R. (1990). A combinatorial unknot detector. Blackburn College research report.

Appendix -- Sample Knot Diagrams







****
STACK ASSEMBLY-- continued from page 44

d. Subscripts.

e. If, else.

f. While loops.

g. User written subroutine and function calls with parameter passing.

Some readers will undoubtedly be asking: "Single or multiple subscripts? Can parameter lists contain any expression or only simple variables?" and othe similar questions. I encourage the students to do as much as each is capable of. Like any good MAJOR project, and this compiler is a major project for these students, my assignment has an open-ended quality about it.

The students are graded on how much they complete. Roughly speaking, completing arithmetic replacement statements yields a C, completing subscripts, ifs and while-loops is a B, and completing subroutine and function calls constitutes an A.

A few words should be added about the description of the target language. My practice is to give them a context-free grammar which describes the language but *which is not suitable without alteration* for writing a recursive descent compiler. My experiences with this have indicated to me that this is exactly right for my students. By giving them a correct description of the target language which however must be left factored and which requires elimination of left recursion they are introduced both to the power of context free grammars and to the fact that they are not a magic solution to all compiling problems. (Of course discussion of context sensitive issues such as variable declaration brings this idea home too.) The students are thus forced to work with context free grammars and to do some of the work of producing a grammar for use in a recursive descent compiler.

CSC '91
Exhibits and Job Fair Information

The ACM Computer Science Conference, which will be held in San Antonio, Texas from March 5-7, 1991 (with the SIGCSE Technical Symposium on March 7-8), is offering a special Academic Rate for college and university exhibitors. A booth in the Exhibits Area provides a university with an excellent means to contact potential graduate students, faculty, and industrial partners. For further information, please contact:

Barbara Corbett
Assistant to the President
Robert T. Kenworthy, Inc.
866 United Nations Plaza
New York NY 10017
(212) 752-0911

Donald J. Bagert, Jr.
Local Exhibits Chairman, CSC '91
Department of Computer Science, Mail Stop 3104
Texas Tech University
Lubbock TX 79409-3104
(806) 742-1189
bedjb@ttacs1.ttu.edu

CSC '91 is a DoD-approved conference.