LISP Program Modification Exercises

The following exercises are in approximate order of difficulty. Students wishing to achieve an "A" grade should attempt most of these exercises. These exercises constitute the bulk of homework and test assignments for the entire course. Most of these exercises ask you to extend the examples. As well as the exercises below, write your own examples. Explore!

All students: Load and run

1. Eliza
2. the recursive Unification algorithm
3. GPS for Blocks World
4. Parcil
5. Streams and filters
6. Logic Programming
7. Object-oriented
8. Parse

The Programs

1. **Eliza**, by Joseph Weizenbaum, transcribed by Peter Norvig.
A famous interactive dialog program, in the form of a Rogerian counselor. Contains a pattern-matcher. Consider how to extend the dialog manager.

2. **Unification and Search**, by Mark Kantrowitz, CMU.
Unification is an extended form of pattern-matching which is integral to inferences engines and query languages. Recursive and iterative versions are included for comparison. read the code, do not worry about the details. The search package is pedagogical, intended to show the similarities across all search algorithms. Although the code is complete, the knowledge representation is missing, so do not expect to run the entire code sample.

3. **GPS**, by Newell and Simon, CMU. Transcribed by Peter Norvig.
The first, famous General Problem Solver. Runs blocks world, so get familiar with the code. Don't forget to use debug mode.

4. **Parcil**, by Erann Gat, JPL.
A C syntax to LISP code recursive descent parser. Not industrial strength.

The next four programs focus on abstraction in coding. They are described in detail in another handout.

5. **Filters and Streams**, by Lugar and Stubblefield, from ideas by Curtis, Waters, and Steele.
Fundamentals of dataflow programming style.

6. **Logic Programming**, by Lugar and Stubblefield, with examples by Bricken.
Fundamentals of declarative logic programming style; an example of meta-linguistic abstraction.

7. **Object-oriented Programming**, by Lugar and Stubblefield.
Fundamentals of object-oriented programming style.

8. **Parse**, by students in Stanford LISP classes.
A recursive descent parser (similar to Parcil), for semi-natural English sentences.

Exercises

1. **Eliza**:

Extend the Eliza program to talk about your favorite subject.

2. **Unify**:

Describe the difference between
- a. pattern matching for equality
- b. pattern matching with variables
- c. unification

3. **GPS**:

Extend the blocks world example to
- a. use 4, 5, and 6 blocks
- b. have limited table space (the trick is to name blocks which must stay on the table, and disallow moving to the table)
- c. do the Towers of Hanoi puzzle
- d. use other representations, such as an explicit Hand, an OnTable predicate, a ClearTop predicate.
- e. Fix the Sussman anomaly.
- f. Use boundary block representations (or others of your choice).

4. **Search**:

Write the missing generic-search functions for Optimal Paths.

5. **(harder) GPS**:

Modify GPS to use smarter search strategies, such as Hill-climbing, Best-first, or Branch-and-Bound.

6. **(harder) Search**:

Use the Kantrowitz generic-search functions to solve Blocks World by providing the knowledge representation as functions for initial-state, goal-p, and children. Evaluate which method is best, either by using the function TIME, or by counting the calls to TRACE.

Use generic-search or GPS to search the state space of some other problem domain, such as Missionaries-and-Cannibals, Sliding-Tiles, Tic-Tac-Toe, Cryptoarithmetic, or N-Queens.

7. (harder) Parcil:

Test Parcil on some real but restricted C code. Extend it to include some high-level C constructs.

8. Streams:

A. Write a stream/filter program to generate the first N prime numbers.

B. (harder) If you are fluent in another programming language, write the same function in that language and compare the ease of writing, the maintainability/extendibility and modularity, and the readability of each version. Write your other language program so that it handles an *arbitrary* N.

C. Change your program(s) to find the first N prime numbers with two occurrences of the digit 9.

D. Change your program(s) to find the first N prime Fibonacci numbers.

9. Logic Programming:

A. Substitute the normal FIRST/REST/CONS/EMPTYP functions for the (simulated) stream functions in the logic code. That is, expand the simulated stream abstraction inline to remove it. Does this improve readability for you?

B. Look at the problem of people liking themselves in the liking examples. Is there a better fix than the one proposed? Write some other liking rules which expose this problem.

C. Add some other facts (not rules) to the three animal databases to exercise the rules which are not used when asking questions about zeke, fred, and tony. What would you do if you wanted to know about types of animals (the animal taxonomy) but not about a particular named animal?

D. There is a major problem with the INFER function. What is it? Simplify the (first, smaller) addition database by removing commutative facts, and add the commutative rule:

```
(rule if ((var a) + (var b) = (var c) (var d))
      then
      ((var b) + (var a) = (var c) (var d)))
```

Figure out what is happening. (harder) Can you fix it?

E. Ask a question about ancestors in the relatives database and figure out what is happening. Ancestor is an example of a transitive relation. (This is related to question D.)

F. Write some other relative rules like GRANDMOTHER, GRANDPARENT, UNCLE.

G. Fix the sibling rule so that a person is not their own sibling.

H. Why is the fact

```
(1 + 0 = 0 1)
```

"out-of-order" in the first addition database? Organize the order of the facts and rules for maximum efficiency.

I. Write an addition rule which abstracts the carry operation out of the database. The trick is to use a carry-flag `((var x) = 1)` which unifies with the fact `(1 = 1)`.

J. (harder) Design an addition rulebase which handles addition of numbers of arbitrary magnitude.

K. (harder) Write a rulebase which solves Cryptoarithms (e.g. SEND+MORE=MONEY). The essence is to add rule(s) which enforce different numbers to be associated with each different letter. Then you might need rules to improve the efficiency of the solution process.

L. (longer but not harder) Write a rulebase for doing multiplication. For doing elementary algebra. For doing differential calculus.

10. Object-oriented Programming:

A. Get other object-oriented examples (from OO textbooks perhaps) and build them in the simple LISP system.

B. (harder) Figure out what you would need to do to include multiple inheritance.

11. Language Parsing:

A. Extend the grammar to include adverbs and pronouns. This will be (harder) if you have difficulty following the lambda forms in the code.

B. Incorporate PARSE-FIND into the code in place of FIND-* and FIND-?.

C. (harder) Write GENERIC-PARSE.

12. Miscellaneous:

A. Write at least five different versions of REVERSE. (harder) Write fifteen significantly different implementations.

B. (harder) Write a semantic net traversal algorithm. This is not hard if you use either the Logic Programming code or the Object-oriented code as a base.

C. Implement the simple robot-in-maze problem. Add the suggested extensions. When you get to adding the Wumpus, it gets (harder).