ACSL Training: Lisp

Sanjit Bhat Alexander Sun

January 13, 2019

1 Fun Facts

- Lisp was developed in the 50's by John McCarthy at MIT. He was a highly influential figure in Artificial Intelligence who later went on to win the Turing Award and found the Stanford AI Lab. Owing to its founder, Lisp was predominently used in AI research.
- After Fortran, Lisp is the second-oldest high-level programming language.
- Ever heard of tree data structures, dynamic typing (not specifying variable types), conditionals, or recursion? Yup, these concepts were pioneered in Lisp.

2 Background Info

First, I recommend spending some time on the Wikipedia page to get a background on Lisp-specific syntax. It delves into why certain things are the way they are.[1]

Next, check out the ACSL wiki for Lisp. Like the name implies, the fundamental idea behind Lisp is that every line of code, from function calls to print statements, is placed inside a linked-list. For example, (function arg1 arg2 arg3 ... argn) calls the function with the subsequent arguments as its input. The highly regular parentheses-based syntax makes the language easy to process. In fact, owing to its regularity, Lisp was one of the first languages that allowed recursion—each of the arguments can themselves be functions.t

Here's the relevant terminology:

1. *atom*: an individual list element that is not a list itself
2. *literal*: **SUPER IMPORTANT**. Defined with a single leading quote. Tells Lisp whether or not to evaluate a statement. Note: watch for the quote whenever you read a problem.
3. *NIL*: Both an atom and a list with no elements. The equivalent of null in Java and NONE in Python. In true or false functions, NIL represents false.

[1] Did you know that Lisp uses prefix notation in its S-expressions? That's why prefix/postfix/infix is an ACSL topic

2.1 Basic Functions

There are 4 basic functions: SET, SETQ, EVAL, and ATOM. (SET arg1 arg2) assigns arg2 to arg1, evaluating the arguments as usual **if they are non-literals**. (SETQ arg1 arg2) is the exact same as SET except that it automatically makes arg1 a literal. Thus, Lisp assumes that arg1 is a variable name.

EVAL resolves arguments, while ATOM tests whether an item is an atom or a list. To test your knowledge, what will each of the following ATOMs return in the following problem? Also, what will (EVAL c) return?

1. (SETQ a '(MULT 2 3))
2. (SET 'b 'a)
3. (SET 'c a)
4. (ATOM b)
5. (ATOM c)

Answer: a is assigned to be the list (MULT 2 3). If there was no quote before (MULT 2 3), Lisp would evaluate arg2 and set a to be 6. b is set to be the literal 'a', which makes it a character and makes the ATOM function return 'true'. c is set to be the variable a, which makes it a list and makes the ATOM function return NIL. Eval c would evaluate the list, returning the value 6.

2.2 Arithmetic Functions

There are 9 main arithmetic functions: ADD, SUB, MULT, DIV, SQUARE, EXP, EQ, POS, NEG. Every function corresponds to its relevant mathematical operation. A few things to note:

- ADD and MULT work on an indefinite number of arguments
- EQ tests for equality, returning "true" or NIL
- POS and NEG test whether their arguments are positive or negative
- ADD, SUB, MULT, and DIV can be written with +, -, *, and /. If you use these symbols, remember to use **prefix** notation.

2.3 List Functions

There are 4 main list functions: CAR, CDR, CONS, REVERSE. Understanding how these work is arguably the hardest part of ACSL Lisp.

(CAR x) returns the first element of list x, and (CDR x)—the "cutter" function—returns x without its first element. For quick access to specific elements in a list, Lisp uses functions like CAADDDAR, where the A's and D's represent successive CAR and CDR operations. For instance, the preceding functions represents (CAR(CAR(CAD(CAD(CAD(CAR x)))))).

CONS(x y) is slightly more complex. It requires y to be a list (x can be anything) and returns the list (x y). Note, if both x and y were lists, the elements of y would be unpacked from the list, but the elements of x would not. E.g., (CONS '(hello idealab) '(from sanjit)) would return ((hello

idealab) from sanjit), not (hello idealab from sanjit). Finally, (REVERSE arg1 arg2 ... argn) returns (argn ... arg2 arg1).

What value does the following code produce:

1. (SETQ z (CONS '(red white blue) (CDR '(THIS is a list))))
2. (CDDAR z)

Answer: the first line would set z to be ((red white blue) is a list). The second line would perform (CDR (CDR (CAR z))), which is (blue).

Critical thinking question: in the previous code, what would happen if we did (CONS (red white blue)) instead of (CONS '(red white blue)). Answer: Lisp would throw an error. Since we omitted the quote, it would try to evaluate (red white blue), assuming red is a function and white and blue are red's arguments. Since the function red is not previously defined, Lisp throws an error.

2.4 Defining Your Own Function

You can define your own function using the DEF function. The syntax is (DEF function_name (args) (expression operating on args)). For example, (DEF second (args) (CAR (CDR args))) creates a function second that returns the second argument of its input.

3 Exercises

1. Evaluate (MULT (ADD 6 5 0) (MULT 5 1 2 2) (DIV 6 (SUB 2 5)))

2. Evaluate (CDR '((2 (3))(4 (5 6) 7)))

3. (SETQ X '(RI VA FL CA TX))
   (CAR (CDR (REVERSE X)))
   What is the value of the CAR expression?

4. Evaluate (CAR '((2 (3)) (4 (5 6)) 7))

5. (SETQ X '(((a (b c) d) e) ((b (c (d e) b)) a)(a b c)((e d) b (a b)(c e d))))
   Evaluate the following expression:
   (CDR (CDR (CDR (REVERSE (CAR (REVERSE (CDR x)))))))

   Note: to accurately evaluate the expression, you need to first understand the structure of how the list is packed. Do this by drawing arrows between corresponding parentheses.

6. Given the function definitions for HY and FY as follows:
   (DEF HY(PARMS)(REVERSE (CDR PARMS)))
   (DEF FY(PARMS)(CAR (HY (CDR PARMS))))
   What is the value of the following?
   (FY '(DO RE (MI FA) SO))

7. Evaluate (EXP (MULT 2 (SUB 5 (DIV (ADD 5 3 4) 2)) 3) 3)

4 Solutions

1. -440
2. ((4 (5 6) 7))
3. CA
4. (2 (3))
5. ((e d))
6. SO
7. -216