Authors: Charles Vath, Sowndarya Krishnan N, Zak Khitirishvili, Rui Zhang

Project 0: Lisp Introduction and Set Operations

CSCI 561

September 14, 2024

1. Describe the Lisp development environment that you used for this project. (*Hint: The correct answer is ALIVE, SLIME, or similar.*)

   We used alive in VsCode.

2. What are the types of the following Lisp expressions?

   (a) `1` : *Bit*

   (b) `(+ 1 2)` : *Integer*

   (c) `'(+ 1 2)` : *List*

   (d) `(eval '(+ 1 2))` : *Integer*

   (e) `(lambda () (+ 1 2))` : *Function*

   (f) `"foo"` : *String*

   (g) `'bar` : *Symbol*

3. Tail Calls:

   (a) What is tail recursion?

   Answer: Tail recursion occurs when the recursive call is the function's final operation, eliminating the need to track previous states or perform post-return operations. This allows compilers or interpreters to optimize these functions into iterative loops that do not expand the call stack, enhancing performance and preventing overflows.
   GeeksforGeeks: Tail Recursion

   (b) In the recursive implementation, will `fold-left` or `fold-right` be more memory-efficient? Why?

   Answer: One would expect fold left to be more memory efficient than fold right. This is because fold right does not use tail recursion while fold left often does, such as in the implementation in the OCaml book linked. The tail recursion in fold left allows for the fold left to not have to add stack frames for every function call. Fold right is unable to use tail recursion because the operation applied in fold right requires an element and the remaining fold of the right side of the list of the element to be computed. This is contrasted with fold left where the first element can be folded and then the function can precede to the rest of the list.
   4.4. Fold - OCaml Programming

4. Lisp and Python represent code differently.

   (a) Contrast the representations of Lisp code and Python code.

   Answer: Lisp represents commands with S-expressions whereas Python uses operators between operands for many commands and involves and allows syntactic sugar. Python syntax especially is known to be easy to understand and this may lessen the burden of documentation to explain code. Even for people unfamiliar with Python, they may be able to understand the function of the code. A cost of the ease of reading

Python code is the interpreter has to interpret the syntax as opposed to just being easily able to form a tree of operations like in Lisp. Lisp syntax however could be cryptic to one untrained to how it works, and having to track where the parentheses are and what they correspond to can be an impediment. However, the parentheses can also make the exact nature of the operations clear if one understands S-expressions and where the parentheses are. It also allows compilers for Lisp to more easily parse the program.

Lisp - Wikipedia

Python & Lisp

(b) How does Python's `eval()` differ from the approach of Lisp?

*A*nswer: Lisp only has expressions and not statements, which means there need not be an analogous exec like in Python. Another difference is that eval in Lisp can leverage that the input list is already in a tree form, which can make it easier to parse for the compiler or interpreter running eval. Python in comparison either uses strings or special 'code' objects as input to eval. Python allows one to specify a restricted environment for eval which can reduce concerns about security risks associated with eval, though apprently some versions of Lisp allow similar functionality as well. Python also evaluates the string or 'code' object as a condition list to be precise according to Python documentation. Lisp allows for even more flexibility however with the ability to selectively evaluate parts of a list otherwise specified to not evaluate. Lisp allows such with backquotes, the '`' character, which can be prepended before a list to tell the program not to evaluate the list except for the parts where a ',' or ',@' is encountered. ',' characters allow for evaluating an expression and returning the result into the corresponding cell of the exterior S-expression with the cdr of that cell remaining as is. ',@' characters allow for evaluating an expression and inserting the result in place of the cell, such that there is no cdr present (unless calling (list _) on it to instead reformat the result as a list with the result in car and cdr pointing back to the next part of the exterior S-expression). To do something like such with Python could be cumbersome due to less emphasis on code as data and corresponding ability to switch between evaluating expressions or keeping them in a certain form.

Lisp - Wikipedia

eval - Wikipedia

eval - Python Documentation

Lecture-02b-Lisp-Handout.pdf - CSCI 561

5. GCC supports an extension to the C language that allows local/nested functions (functions contained in other functions). A GCC local function can access local variables from its parent function.

   (a) What problems would arise if you return a function pointer to a GCC local function? *(Hint: "Funarg problem")*

   *A*nswer: Returning a function pointer to a GCC local function introduces the "funarg problem." This issue arises because local functions can access and modify local variables of their parent functions. However, once the parent function exits, its local variables go out of scope and may be deallocated, leading to dangling references in the nested function. If the function pointer is called after its enclosing scope has been destroyed, it may attempt to access these invalid memory locations, causing undefined behavior or crashes.

   Nested Functions

   (b) How does Lisp handle this problem?

   *A*nswer: Lisp handles this problem through the use of closures. A closure in Lisp is a function that captures the local variables from its environment when it is defined. This means that even after the outer function has returned, the local variables are still accessible to the nested function, preserved as part of the closure's environment. This avoids the issues of dangling references or undefined behavior as seen in languages without proper closure support. Lisp's runtime automatically manages the lifecycle of these variables, ensuring they remain valid as long as the closure exists and can potentially access them.

   Closure-Wiki(Applications section)

6. Test the performance of your implementation of `merge-sort`.

(a) Plot the running time of both your `merge-sort` implementation and the builtin Lisp `sort` function for increasing input sizes. Include enough data points to demonstrate the empirical asymptotic running time.

*A*nswer: We timed both functions and ran 8 samples, which we averaged to get the results shown below in Figure 1.

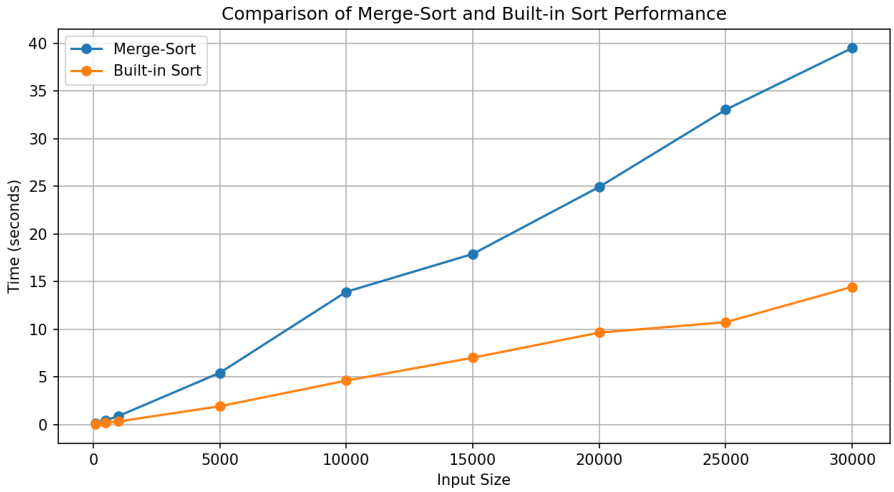

Figure 1: runtime vs input size

(b) What asymptotic running time did you expect for `merge-sort` and the builtin Lisp `sort` function, and what running time did you observe? Explain any differences.

*A*nswer: Both the merge-sort implementation and the built-in Lisp sort function have an expected asymptotic running time of $O(n \log n)$. In the observed results, both algorithms followed this trend, but the built-in sort function was consistently faster. This is likely due to low-level optimizations, such as better memory management and cache utilization, and possibly using an adaptive sorting algorithms. In contrast, the merge-sort implementation has more overhead due to recursive function calls and list manipulations in Lisp (subseq, car, cdr ...), which are not as optimized as the array-based operations used by the built-in sort. While both algorithms scale similarly, the built-in sort outperforms my merge-sort.

7. Newton's method is a powerful technique for finding roots of real-valued functions. Given a function $p(x)$, Newton's method iteratively computes the next approximation x_{n+1} of the root using its derivative, $p'(x)$, as follows:

$$x_{n+1} = x_n - \frac{p(x_n)}{p'(x_n)}$$

For this question, you will use your implementation of `find-fixpoint` to find roots of polynomials via Newton's method.

(a) Determine $f(x)$:

Given a polynomial $p(x)$ and its derivative $p'(x)$, determine the function $f(x)$ that should be used in the `find-fixpoint` function. Write down the expression for $f(x)$.

Answer:

$$f(x) = x - \frac{p(x)}{p'(x)}$$

(b) Implement and Test:

Implement the function $f(x)$ you derived, and use `find-fixpoint` to find the roots of the following polynomials using Newton's method:

i. $p(x) = x^2 - 2$
ii. $p(x) = x^3 - x - 2$
iii. $p(x) = x^3 - 6x^2 + 11x - 6$ (roots are 1, 2, 3)

For each polynomial, choose an initial guess and use a precision of 0.001. Give the parameters to the call to `find-fixpoint` and report the root found by your implementation.

Answer:

Note: I experimented with several initial guesses for each polynomial and selected the ones that led to the correct roots. These initial guesses were chosen based on the following criteria:

- For $p(x) = x^2 - 2$, an initial guess of 1.0 was chosen because the root $\sqrt{2}$ is approximately 1.414, and starting close to this value ensures convergence.
- For $p(x) = x^3 - x - 2$, the initial guess of 2.5 was selected as it is close to the expected root, which lies between 1 and 2.
- For $p(x) = x^3 - 6x^2 + 11x - 6$, I used initial guesses around 0.8, 2.1, and 2.9 to find the roots at 1, 2, and 3 respectively. These values were chosen based on the fact that the roots of this polynomial are known, and starting near each root ensured quick convergence.
- **Polynomial 1**: $p(x) = x^2 - 2$
  - Initial guess: 1.0
  - Function call: `(find-fixpoint #'newton-f1 1.0 (lambda (a b) (< (abs (- a b)) 0.001)) 6)`
  - Root found: 1.414214 (which is $\sqrt{2}$)
- **Polynomial 2**: $p(x) = x^3 - x - 2$
  - Initial guess: 2.5
  - Function call: `(find-fixpoint #'newton-f2 2.5 (lambda (a b) (< (abs (- a b)) 0.001)) 6)`
  - Root found: 1.521380
- **Polynomial 3**: $p(x) = x^3 - 6x^2 + 11x - 6$
  - Initial guess for $x = 1$: 0.8
  - Function call: `(find-fixpoint #'newton-f3 0.8 (lambda (a b) (< (abs (- a b)) 0.001)) 6)`
  - Root found: 1.0
  - Initial guess for $x = 2$: 2.1

- Function call: `(find-fixpoint #'newton-f3 2.1 (lambda (a b) (< (abs (- a b)) 0.001)) 6)`
- Root found: 1.999999
- Initial guess for $x = 3$: 2.9
- Function call: `(find-fixpoint #'newton-f3 2.9 (lambda (a b) (< (abs (- a b)) 0.001)) 6)`
- Root found: 3.0