**Reflection: Development of Racket Expression Evaluator**

**Hashim Abdulla**

**Due date: 5/4/2025**

***Overview of the Project***

For this assignment, I needed to modify an existing Racket implementation of a numeric expression evaluator. The original code used a Maybe monad for handling potential errors, and I had to convert it to an Either (Result) pattern, add variable management with state, and implement a REPL interface. This required understanding functional programming concepts like monads, state management without mutation, and Racket's specific syntax and libraries.

***LLM Tools I Used***

***Initial Attempts: GPT-4o and Gemini 2.5 Pro***

My first approach was to use OpenAI's GPT-4o model, which I had previous experience with for other programming assignments. However, I encountered significant issues:

* The generated Racket code contained numerous syntax errors
* The logical implementation of the Either monad was fundamentally flawed
* The state management approach didn't follow functional programming principles
* Many of the suggested functions didn't actually work in Racket for some reason
* The model seemed to confuse Racket with other Lisp dialects at times

I then tried Google's Gemini 2.5 Pro, hoping for better Racket support, but encountered similar issues:

* The code it produced had even more basic syntax errors
* The implementation of state management wasn't thread-safe through recursion
* It often generated solutions that were closer to imperative programming than functional
* When asked to fix issues, it often introduced new problems

I spent hours trying to debug these issues, researching on Reddit and Stack Overflow, but many of the problems were so fundamental that fixing them would require essentially rewriting the entire solution.

***Success with Claude 3.7 Sonnet***

After these frustrating experiences, I went back to Anthropic's Claude 3.7 Sonnet, which I’ve had success with in the past:

* The Racket code was syntactically correct from the first attempt for every prompt
* The implementation properly followed functional programming principles
* The Either monad was correctly implemented with success/failure types
* The state management approach properly preserved immutability
* Error handling was also good and followed the specification exactly

With Claude, I was able to get a working implementation that I could understand and that correctly handled all the test cases.

***Where LLMs Were Helpful***

1. **Understanding functional concepts**: Claude was excellent at explaining the Either monad pattern and how it differs from Maybe.
2. **Converting between paradigms**: The LLM helped translate the Maybe-based error handling to the Either approach smoothly.
3. **Implementing complex state management**: The functional approach to state (passing and returning state) was implemented correctly.
4. **Error handling**: Claude generated comprehensive error handling for all edge cases specified in the requirements.
5. **Code structure**: All of the LLMs’ overall structure was clean, well-commented, and followed good functional programming practices for the most part.

***Where LLMs Fell Short***

1. **Language-specific knowledge**: GPT-4o and Gemini struggled with Racket-specific syntax and libraries.
2. **Debugging capabilities**: When code had errors, the earlier models often couldn't diagnose the real issues.
3. **Consistency**: The non-Claude models would sometimes contradict themselves or forget parts of the implementation.
4. **Understanding functional programming**: Only Claude seemed to truly understand the functional programming paradigm required.

***Impact on My Programming Skills***

***Positive Impacts***

* **Exposure to different approaches**: Seeing different implementations helped me understand the problem from multiple angles.
* **Learning functional patterns**: Working with LLM-generated functional code has improved my understanding of concepts like monads and state management.
* **Faster prototyping**: LLMs can quickly generate working code structures that I can then modify and improve.
* **Better commenting and documentation**: The well-commented code from Claude has improved my own documentation practices.

***Potential Hindrances***

* **Dependency risk**: I worry about becoming too dependent on LLMs for solving problems instead of developing my own problem-solving skills.
* **Understanding gaps**: Sometimes I found myself implementing solutions I didn't fully understand, which isn't good for long-term learning.
* **False confidence**: LLMs can generate code that looks correct but contains subtle logical errors that are hard to detect.
* **Reduced practice**: I feel like less time spent struggling with syntax means potentially less internalization of language features.

***Skills Needed for Effective LLM Use***

1. **Critical code review ability**: Being able to analyze and verify generated code is essential.
2. **Understanding in programming concepts**: Understanding at least basic principles helps identify when LLM suggestions are incorrect.
3. **Prompt engineering**: Learning how to effectively communicate requirements to LLMs definitely makes a huge difference in output quality.
4. **Testing strategies**: Running tests to verify LLM-generated code is crucial.
5. **LLM model awareness**: Understanding the strengths and weaknesses of different models helps choose the right tool.

***Big Picture Takeaways***

1. **LLMs as programming partners, not replacements**: The most effective use is collaborative, where I guide the LLM and review its work.
2. **Model selection matters significantly**: The difference between Claude and the other models for this task was dramatic.
3. **Domain-specific knowledge is still crucial**: My understanding of functional programming was essential for evaluating the solutions.
4. **"Trust but verify"**: Always reviewing and testing LLM code is non-negotiable.

***Advice for Future Students and Faculty***

***For Students:***

1. **Use multiple models**: Different LLMs have different strengths; don't rely on just one.
2. **Learn to prompt effectively**: The quality of your prompts directly affects the quality of the generated code.
3. **Always understand the code**: Never submit code you don't understand, even if it works.
4. **Use LLMs to learn, not just to complete assignments**: Ask for explanations, not just solutions.
5. **Maintain your fundamental skills**: Continue practicing coding without LLMs regularly.

***For Faculty:***

1. **Design LLM-resistant assignments**: Create problems that require deeper understanding and creativity.
2. **Teach prompt engineering**: Include instruction on how to effectively work with LLMs.
3. **Focus on understanding, not just implementation**: Emphasize explanation and justification of code.
4. **Embrace LLMs as tools**: Design curriculum that teaches how to effectively use these tools rather than trying to work around them.
5. **Balance traditional and LLM-assisted learning**: Some assignments should still be done without LLM assistance to build fundamental skills.

***Conclusion***

My experience developing this Racket expression evaluator has shown me both the potential and limitations of current LLM technology for programming tasks. Both GPT-4o and Gemini 2.5 Pro struggled with the functional programming aspects and Racket-specific details, but Claude 3.7 Sonnet proved to be an exceptional tool that produced clean, correct code that followed the specification exactly.

These tools are transforming how we approach programming education and practice. When used thoughtfully—as assistants rather than replacements for our own understanding—they can accelerate learning and implementation. Albeit, they still require solid programming foundations and critical thinking skills to be used effectively.

As we continue into this new era of AI-assisted programming, developing a balanced approach that leverages these tools while maintaining our core skills will be essential for students and professionals alike.