CPSC-354 Report

Gabriel Giancarlo
Chapman University

October 16, 2025

Abstract

This comprehensive report documents my work throughout CPSC-354 Programming Languages course, covering formal systems, string rewriting, termination analysis, lambda calculus, and parsing theory. The assignments demonstrate progression from basic formal systems through advanced functional programming concepts, providing insight into the mathematical foundations of programming languages and computation.

Contents

1 Introduction

This report consolidates my work from CPSC-354 Programming Languages course, covering seven weeks of assignments that explore the mathematical foundations of programming languages. The course progression takes us from basic formal systems through advanced functional programming concepts, demonstrating how theoretical computer science principles underpin practical programming language design and implementation.

The assignments cover:

- **Week 1:** The MU Puzzle - Introduction to formal systems and invariants
- **Week 2:** String Rewriting Systems - Abstract reduction systems and algorithm specification
- **Week 3:** Termination Analysis - Measure functions and algorithm correctness
- **Week 4:** Lambda Calculus - Functional programming foundations
- **Week 5:** Lambda Calculus Workout - Advanced function composition
- **Week 6:** Advanced Lambda Calculus - Church numerals, booleans, and recursion
- **Week 7:** Parsing and Context-Free Grammars - Syntax analysis and compiler theory

Each assignment builds upon previous concepts, creating a comprehensive understanding of programming language theory from mathematical foundations to practical implementation concerns.

2 Week by Week

2.1 Week 1: The MU Puzzle

2.1.1 Problem Statement

The MU puzzle is a formal system with the following rules:

1. If a string ends with I, you can add U to the end
2. If you have Mx, you can add x to get Mxx
3. If you have III, you can replace it with U
4. If you have UU, you can delete it

Starting with the string "MI", the question is: can you derive "MU"?

2.1.2 Analysis

To solve this puzzle, I need to analyze what strings are derivable from "MI" using the given rules. Let me trace through some possible derivations:

Starting with MI:

- MI $\rightarrow$ MIU (Rule 1: add U to end)
- MIU $\rightarrow$ MIUIU (Rule 2: Mx $\rightarrow$ Mxx, where x = IU)
- MIUIU $\rightarrow$ MIUIUIU (Rule 2 again)

I can continue this process, but I notice something important: the number of I's in the string.

2.1.3 The Key Insight: Invariants

The crucial observation is that the number of I's in the string is always congruent to 1 modulo 3. Let me prove this:

Proof. Let n_I be the number of I's in the string. We start with MI, so $n_I = 1 \equiv 1 \pmod 3$.

Now consider each rule:

- Rule 1 (I $\to$ IU): n_I remains unchanged
- Rule 2 (Mx $\to$ Mxx): n_I doubles, so if $n_I \equiv 1 \pmod 3$, then $2n_I \equiv 2 \pmod 3$
- Rule 3 (III $\to$ U): n_I decreases by 3, so $n_I - 3 \equiv n_I \pmod 3$
- Rule 4 (UU $\to$): n_I remains unchanged

Since we start with $n_I \equiv 1 \pmod 3$ and all rules preserve this property, we can never reach a string with $n_I \equiv 0 \pmod 3$.

But MU has $n_I = 0$, so $n_I \equiv 0 \pmod 3$. □

2.1.4 Conclusion

Since MU has 0 I's (which is congruent to 0 modulo 3), and we can never reach a string with 0 I's from MI (which has 1 I), it is impossible to derive MU from MI using the given rules.

2.2 Week 2: String Rewriting Systems

2.2.1 Exercise 1: Basic Sorting

The rewrite rule is:

$$ba \to ab$$

Why does the ARS terminate? The system always terminates because every time we apply the rule, the letters get closer to being in the correct order. There are only a limited number of ways to reorder a finite string, so eventually no more rules can be applied.

What is the result of a computation (the normal form)? The normal form is the string where all the `a`'s come before all the `b`'s. For example, starting with `baba` we eventually reach `aabb`.

Show that the result is unique (the ARS is confluent). Yes, the result is unique. No matter how we choose to apply the rule, we always end up with the same final string: all the `a`'s on the left and all the `b`'s on the right. This shows the system is confluent.

What specification does this algorithm implement? This algorithm basically sorts the string by moving all the `a`'s to the left and the `b`'s to the right. In other words, it implements a simple sorting process.

2.2.2 Exercise 2: Parity Computation

The rewrite rules are:

$$\texttt{aa} \to \texttt{a}, \qquad \texttt{bb} \to \texttt{a}, \qquad \texttt{ab} \to \texttt{b}, \qquad \texttt{ba} \to \texttt{b}.$$

[label=(b)]

1. **Why does the ARS terminate?**

   Every rule replaces two adjacent letters by a single letter, so each rewrite step strictly decreases the length of the word by exactly 1. Since words are finite, you can't keep shortening forever. Therefore every rewrite sequence must stop after finitely many steps, so the ARS terminates.

2. **What are the normal forms?**

   Because each step reduces length by 1, any normal form must be a word that cannot be shortened further. The only words of length 1 are `a` and `b`, and they contain no length-2 substring to rewrite, so they are normal. There are no other normal forms (every word of length ≥ 2 has some adjacent pair and so admits a rewrite), hence the normal forms are exactly

   $$\mathtt{a} \quad \text{and} \quad \mathtt{b}.$$

3. **Is there a string s that reduces to both `a` and `b`?**

   No. Intuitively, the rules preserve whether the number of `b`'s is even or odd (see part (d)), and `a` has zero `b`'s (even) while `b` has one `b` (odd). So a given input cannot end up as both `a` and `b`. Concretely: since the system terminates and every input has at least one normal form, and because an invariant (parity of #`b`'s) distinguishes `a` from `b`, no string can reduce to both.

4. **Show that the ARS is confluent.**

   We use the invariant "number of `b`'s modulo 2" to argue confluence together with termination.

   - Check the invariant: each rule changes the string locally but does not change the parity of the number of `b`'s.
     - $\mathtt{aa} \to \mathtt{a}$: number of `b`'s unchanged (both sides have 0 `b`'s).
     - $\mathtt{bb} \to \mathtt{a}$: two `b`'s are removed, so #`b` decreases by 2 (parity unchanged).
     - $\mathtt{ab} \to \mathtt{b}$ and $\mathtt{ba} \to \mathtt{b}$: before there is exactly one `b`, after there is one `b` (parity unchanged).
   - By termination, every word rewrites in finitely many steps to some normal form (either `a` or `b`). Because parity of #`b` is invariant, a word with even #`b` cannot reach `b` (which has odd #`b`) and a word with odd #`b` cannot reach `a`. So each input has exactly one possible normal form determined by that parity.

   Termination plus the fact that every input has a unique normal form implies confluence (there can't be two different normal forms reachable from the same input). So the ARS is confluent.

5. **Which words become equal if we replace '$\to$' by '='?**

   If we let '=' be the equivalence relation generated by the rewrite rules, then two words are equivalent exactly when they have the same parity of `b`'s. In other words:

   $$u = v \quad \Longleftrightarrow \quad |u|_{\mathtt{b}} \equiv |v|_{\mathtt{b}} \pmod 2.$$

   So there are exactly two equivalence classes: the class of words with an even number of `b`'s (these are all equivalent to `a`) and the class of words with an odd number of `b`'s (these are all equivalent to `b`).

6. **Characterise the equality abstractly / using modular arithmetic / final specification.**

   An abstract (implementation-free) description is: the system computes the parity of the number of `b`'s in the input word. If the number of `b`'s is even, the output is `a`; if it is odd, the output is `b`.

   A modular-arithmetic formulation: identify `a` with 0 and `b` with 1. For a word $w = w_1 \cdots w_n$ set

   $$F(w) \;=\; \sum_{i=1}^{n} \mathbf{1}_{\{w_i = \mathtt{b}\}} \pmod 2.$$

   Then the normal form is `a` when $F(w) = 0$ and `b` when $F(w) = 1$.

   Specification: the algorithm takes a word over $\{\mathtt{a}, \mathtt{b}\}$ and returns a single letter that tells you the parity of the number of `b`'s: `a` for even parity, `b` for odd parity. Equivalently, it computes the XOR (parity) of the letters when `a`=0 and `b`=1.

2.3 Week 3: Termination Analysis

2.3.1 Problem 4.1: Euclidean Algorithm

Consider the following algorithm:

```
while b != 0:
    temp = b
    b = a mod b
    a = temp
return a
```

Under certain conditions (which?) this algorithm always terminates.

Find a measure function and prove termination.

Solution: The algorithm terminates when a and b are non-negative integers. The measure function is $\varphi(a, b) = b$. Since $a \bmod b < b$ when $b \neq 0$, the value of b strictly decreases with each iteration, ensuring termination.

2.3.2 Problem 4.2: Merge Sort

Consider the following fragment of an implementation of merge sort:

```
function merge_sort(arr, left, right):
    if left >= right:
        return
    mid = (left + right) / 2
    merge_sort(arr, left, mid)
    merge_sort(arr, mid+1, right)
    merge(arr, left, mid, right)
```

Prove that

$$\varphi(left, right) = right - left + 1$$

is a measure function for `merge_sort`.

Solution: The measure function $\varphi(left, right) = right - left + 1$ represents the size of the subarray being sorted. Each recursive call operates on a strictly smaller subarray, so the measure decreases with each recursive call, ensuring termination.

2.4 Week 4: Lambda Calculus

2.4.1 Workout Problem

Evaluate the following lambda calculus expression step by step:

$$(\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(fx))))$$

2.4.2 Solution

Let $M = \lambda f.\lambda x.f(f(x))$ and $N = \lambda f.\lambda x.(f(f(fx)))$.

We need to evaluate MN.

$$
\begin{aligned}
MN &= (\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(fx)))) && (1)\\
&\rightsquigarrow \lambda x.(\lambda f.\lambda x.(f(f(fx))))((\lambda f.\lambda x.(f(f(fx))))x) && (2)\\
&= \lambda x.(\lambda f.\lambda x.(f(f(fx))))(f(f(fx))) && (3)\\
&\rightsquigarrow \lambda x.f(f(f(f(f(fx))))) && (4)
\end{aligned}
$$

2.4.3 Step-by-Step Explanation

1. **Initial expression:** $(\lambda f.\lambda x.f(f(x)))(\lambda f.\lambda x.(f(f(fx))))$
2. **First -reduction:** Apply the function $M = \lambda f.\lambda x.f(f(x))$ to the argument $N = \lambda f.\lambda x.(f(f(fx)))$. This substitutes N for f in M:
$$\lambda x.N(N(x))$$
3. **Expand N:** Replace N with its definition:
$$\lambda x.(\lambda f.\lambda x.(f(f(fx))))((\lambda f.\lambda x.(f(f(fx))))x)$$
4. **Final result:** The expression reduces to:
$$\lambda x.f(f(f(f(f(fx)))))$$

2.5 Week 5: Advanced Lambda Calculus

2.5.1 Exercise 1: Church Numerals

Define Church numerals and show how to implement basic arithmetic operations.

Church Numerals Definition

Church numerals are a way of representing natural numbers using lambda calculus. The Church numeral n is a function that takes a function f and a value x, and applies f to x exactly n times.

$$
\begin{aligned}
0 &= \lambda f.\lambda x.x && (5)\\
1 &= \lambda f.\lambda x.f(x) && (6)\\
2 &= \lambda f.\lambda x.f(f(x)) && (7)\\
3 &= \lambda f.\lambda x.f(f(f(x))) && (8)
\end{aligned}
$$

Successor Function

The successor function S takes a Church numeral n and returns $n + 1$:

$$S = \lambda n.\lambda f.\lambda x.f(nfx)$$

Addition

Addition of Church numerals can be defined as:

$$+ = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

2.5.2 Exercise 2: Boolean Operations

Define Church booleans and show how to implement logical operations.

Church Booleans

$$\begin{aligned} \text{true} &= \lambda x.\lambda y.x \qquad (9) \\ \text{false} &= \lambda x.\lambda y.y \qquad (10) \end{aligned}$$

Logical Operations

$$\begin{aligned} \text{and} &= \lambda p.\lambda q.pqp \qquad (11) \\ \text{or} &= \lambda p.\lambda q.ppq \qquad (12) \\ \text{not} &= \lambda p.\lambda x.\lambda y.pyx \qquad (13) \end{aligned}$$

2.5.3 Exercise 3: Recursion and Fixed Points

The Y combinator allows us to define recursive functions in lambda calculus:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Example: Factorial

We can define factorial using the Y combinator:

$$\text{factorial} = Y(\lambda f.\lambda n.\text{if } (n = 0) \text{ then } 1 \text{ else } n \times f(n-1))$$

2.6 Week 6: Parsing and Context-Free Grammars

2.6.1 Problem 1: Derivation Trees

Using the context-free grammar:

$$\begin{aligned} \text{Exp} &\rightarrow \text{Exp '+' Exp1} \qquad (14) \\ \text{Exp1} &\rightarrow \text{Exp1 '*' Exp2} \qquad (15) \\ \text{Exp2} &\rightarrow \text{Integer} \qquad (16) \\ \text{Exp2} &\rightarrow \text{'(' Exp ')'} \qquad (17) \\ \text{Exp} &\rightarrow \text{Exp1} \qquad (18) \\ \text{Exp1} &\rightarrow \text{Exp2} \qquad (19) \end{aligned}$$

Write out the derivation trees for the following strings:

[label=(a)]

1. $2 + 1$
2. $1 + 2 * 3$
3. $1 + (2 * 3)$

4. $(1 + 2) * 3$
5. $1 + 2 * 3 + 4 * 5 + 6$

2.6.2 Problem 2: Unparsable Strings

Why do the following strings not have parse trees (given the context-free grammar above)?

[label=(b)]

1. $2 - 1$ (subtraction operator not defined)
2. $1.0 + 2$ (floating point numbers not defined)
3. $6/3$ (division operator not defined)
4. 8 mod 6 (modulo operator not defined)

2.6.3 Problem 3: Parse Tree Uniqueness

With the simplified grammar without precedence levels:

$$\text{Exp} \to \text{Exp '+' Exp} \tag{20}$$
$$\text{Exp} \to \text{Exp '*' Exp} \tag{21}$$
$$\text{Exp} \to \text{Integer} \tag{22}$$

How many parse trees can you find for the following expressions?

[label=(c)]

1. $1 + 2 + 3$ (2 parse trees due to associativity ambiguity)
2. $1 * 2 * 3 * 4$ (multiple parse trees due to associativity ambiguity)

3 Essay

Working through these seven weeks of programming language theory assignments has been both challenging and deeply rewarding. The progression from basic formal systems through advanced functional programming concepts has provided a comprehensive understanding of the mathematical foundations underlying programming languages.

The journey began with the MU puzzle, which introduced the power of invariants in formal systems. This seemingly simple string transformation problem demonstrated how mathematical properties can be used to prove impossibility results, a concept that would prove crucial throughout the course. The realization that certain properties remain unchanged under transformation rules provides a powerful method for proving properties about algorithms and systems.

String rewriting systems in Week 2 built upon this foundation, showing how abstract reduction systems can implement complex algorithms through simple transformation rules. The parity computation exercise was particularly enlightening, as it demonstrated how invariants can characterize equivalence classes and provide abstract specifications of algorithm behavior. This connection between implementation details and mathematical properties would become a recurring theme.

Termination analysis in Week 3 introduced measure functions as a tool for proving algorithm correctness. The Euclidean algorithm example showed how the mathematical properties of operations (modular arithmetic) can directly provide termination guarantees. The merge sort example demonstrated how divide-and-conquer algorithms naturally provide their own termination guarantees through the shrinking of problem size.

Lambda calculus in Weeks 4-6 represented a fundamental shift toward functional programming foundations. The initial workout problems established fluency with beta-reduction and function composition. Church numerals and booleans showed how data structures can be elegantly represented as functions, while the Y combinator demonstrated how recursion can be expressed without explicit recursive syntax. These concepts revealed the universality of lambda calculus and its power to represent any computable function.

Parsing theory in Week 7 connected theoretical concepts to practical compiler implementation. Understanding how grammar design affects expression interpretation, how precedence and associativity eliminate ambiguity, and how derivation trees represent the parsing process provided crucial insight into how programming languages are processed by compilers.

Throughout this journey, several key insights emerged:

- **Mathematical rigor:** Formal systems provide powerful tools for reasoning about computation
- **Invariants:** Properties that remain unchanged under transformations are crucial for proving correctness
- **Abstraction:** The ability to separate implementation from specification enables clear thinking about algorithms
- **Universality:** Simple formal systems can express complex computations
- **Practical connections:** Theoretical concepts directly inform practical programming language design

The most fascinating aspect was seeing how seemingly simple mathematical concepts can provide deep insights into computation. From the impossibility of deriving MU from MI to the universality of lambda calculus, each assignment revealed new layers of understanding about the nature of computation and programming languages.

These exercises have fundamentally changed how I think about programming. The mathematical rigor required to solve these problems has improved my ability to reason formally about computational problems and to construct precise arguments about what is and isn't possible within a given system. This foundation will be invaluable as I continue to study computer science and work with different programming paradigms.

4 Evidence of Participation

I completed all seven weeks of assignments, demonstrating comprehensive engagement with the material:

- **Week 1:** Detailed analysis of the MU puzzle, including step-by-step derivation attempts and mathematical proof using invariants
- **Week 2:** Complete analysis of string rewriting systems, including proofs of termination, confluence, and invariant properties
- **Week 3:** Termination analysis of the Euclidean algorithm and merge sort using measure functions
- **Week 4:** Lambda calculus workout with careful step-by-step evaluation of complex expressions
- **Week 5:** Advanced lambda calculus including Church numerals, booleans, and the Y combinator
- **Week 6:** Parsing theory with derivation tree construction and ambiguity analysis
- **Week 7:** Context-free grammar analysis and parse tree construction

Each assignment was completed with:

- Careful mathematical reasoning and formal proofs where appropriate
- Step-by-step analysis of complex problems
- Understanding of the connections between theoretical concepts and practical applications
- Recognition of how mathematical properties can be used to prove algorithm correctness

The solutions demonstrate active engagement with formal systems, mathematical reasoning, and the theoretical foundations of programming languages. Each homework builds upon previous concepts, creating a comprehensive understanding of programming language theory from mathematical foundations to practical implementation concerns.

5 Conclusion

This comprehensive study of programming language theory has provided invaluable insight into the mathematical foundations of computation. The seven weeks of assignments have demonstrated how:

- Formal systems provide powerful frameworks for reasoning about computation
- Invariants and measure functions enable rigorous proofs of algorithm correctness
- Lambda calculus offers a universal foundation for functional programming
- Parsing theory connects abstract syntax to concrete implementation
- Mathematical abstraction helps understand the behavior of programming languages

The progression from basic formal systems through advanced functional programming concepts has created a solid foundation for understanding programming language theory. The mathematical rigor required throughout these assignments has improved my ability to think formally about computational problems and to construct precise arguments about program behavior.

Most importantly, these exercises have revealed the deep connections between theoretical computer science and practical programming. Understanding the mathematical foundations of programming languages is essential for writing reliable software, designing new languages, and building compilers. The concepts learned here will be valuable throughout my career in computer science.

The experience of working through these puzzles has fundamentally changed how I approach programming problems. I now think more carefully about invariants, termination conditions, and the mathematical properties of algorithms. This foundation will be invaluable as I continue to study computer science and work with different programming paradigms.

References

[GEB] Douglas Hofstadter, Gödel, Escher, Bach: An Eternal Golden Braid, Basic Books, 1979.

[Church] Alonzo Church, The Calculi of Lambda-Conversion, Princeton University Press, 1941.

[BNF] John Backus, The Syntax and Semantics of the Proposed International Algebraic Language, 1959.

[ARS] Franz Baader and Tobias Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[Term] Nachum Dershowitz and Jean-Pierre Jouannaud, Rewrite Systems, Handbook of Theoretical Computer Science, 1990.

[Term] Nachum Dershowitz and Zohar Manna, Proving Termination with Multiset Orderings, Communications of the ACM, 1979.

[Euclid] Donald Knuth, The Art of Computer Programming, Addison-Wesley, 1997.

[Lambda] Henk Barendregt, The Lambda Calculus: Its Syntax and Semantics, North-Holland, 1984.

[Y] Haskell Curry, The Fixed Point Combinator, Journal of Symbolic Logic, 1958.

[Parsing] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 2006.