CPSC-354 Report

Gabriel Giancarlo
Chapman University

October 13, 2025

Abstract

This report documents my work on advanced lambda calculus topics, including Church numerals, boolean operations, and recursion through fixed point combinators. I explored how lambda calculus can represent natural numbers, logical operations, and recursive functions, demonstrating the power and expressiveness of functional programming foundations.

Contents

1 Introduction

This week's assignment continues our exploration of lambda calculus with more advanced topics. We build upon the foundations established in previous assignments to develop deeper understanding of functional programming principles, including how to represent data structures and control flow purely through function application and abstraction.

The key topics covered include:

- Church numerals for representing natural numbers
- Church booleans for representing truth values
- Logical operations using lambda calculus
- Recursion through fixed point combinators

2 Week by Week

2.1 Week 1: Advanced Lambda Calculus

2.1.1 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.

$$0 = \lambda f.\lambda x.x \tag{1}$$
$$1 = \lambda f.\lambda x.f(x) \tag{2}$$
$$2 = \lambda f.\lambda x.f(f(x)) \tag{3}$$
$$3 = \lambda f.\lambda x.f(f(f(x))) \tag{4}$$

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:

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

2.1.2 Exercise 2: Boolean Operations

Define Church booleans and show how to implement logical operations.

Church Booleans

$$\text{true} = \lambda x.\lambda y.x \tag{5}$$
$$\text{false} = \lambda x.\lambda y.y \tag{6}$$

Logical Operations

$$\text{and} = \lambda p.\lambda q.pqp \tag{7}$$
$$\text{or} = \lambda p.\lambda q.ppq \tag{8}$$
$$\text{not} = \lambda p.\lambda x.\lambda y.pyx \tag{9}$$

2.1.3 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:

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

3 Essay

Working through these advanced lambda calculus concepts was both challenging and deeply rewarding. The most fascinating aspect was seeing how we can represent all of computation using nothing but functions.

Church numerals were particularly eye-opening. The idea that we can represent numbers as functions that apply another function a certain number of times is elegant and powerful. It shows how lambda calculus can encode not just computation, but also data structures.

The Church boolean system was equally impressive. Using functions to represent truth values, where `true` selects its first argument and `false` selects its second, is a beautiful example of how lambda calculus can represent logical operations purely through function application.

The fixed point combinator (Y combinator) was the most challenging concept. The idea that we can define recursive functions without explicit recursion syntax is mind-bending. The Y combinator essentially provides a way to "unfold" recursive definitions, allowing us to express any recursive function in pure lambda calculus.

These exercises highlighted several key insights:

- **Data as functions:** Numbers, booleans, and other data can be represented as functions
- **Computation as application:** All computation reduces to function application
- **Recursion without syntax:** The Y combinator enables recursion in a purely functional setting
- **Universality:** Lambda calculus can express any computable function

4 Evidence of Participation

I completed all the advanced lambda calculus exercises, including:

- **Church Numerals:** Defined Church numerals and implemented successor and addition functions
- **Church Booleans:** Defined boolean values and logical operations (and, or, not)
- **Fixed Point Combinator:** Understood how the Y combinator enables recursion
- **Factorial Example:** Worked through defining factorial using the Y combinator

Each exercise was completed with:

- Careful mathematical definitions
- Step-by-step explanations of function behavior
- Understanding of how functions represent data and operations

- Recognition of the power and elegance of functional representation

5 Conclusion

These advanced lambda calculus exercises provided deep insight into the mathematical foundations of functional programming. The key lessons learned include:

- Lambda calculus provides a universal foundation for computation
- Data structures can be elegantly represented as functions
- Recursion can be expressed without explicit recursive syntax
- The Y combinator demonstrates the power of higher-order functions
- Functional programming has deep mathematical foundations

Understanding these advanced concepts is essential for functional programming. These exercises have improved my ability to think about computation in terms of functions and to understand the theoretical foundations that underlie modern functional programming languages.

References

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

[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.