MicroHaskell

Project Report

as part of the course
[redacted]

submitted at
Department of Mathematics, Natural Sciences, and Computer Science
at [redacted]

by
[redacted]

07.09.2025

Student ID	[redacted]
Degree Program	Computer Science, Master
Semester	Summer Semester 2025
Lecturer	[redacted]

Contents

1 Introduction

This report documents the design and implementation of an interpreter for the programming lanuage MicroHaskell, a simplified subset of the Haskell programming language. MicroHaskell retains key features of Haskell's functional paradigm, including first-class functions, lazy-evaluation, and recursive binding, while omitting more complex constructs to facilitate analysis and implementation. It was developed as part of the course "[redacted]" at [redacted] during the summer semester of 2025.

The primary goal of this project is to model the core principles of Haskell within a reduced and well-defined language subset. The interpreter is designed to parse and evaluate MicroHaskell programs in a way that is strongly rooted in the concepts of λ-calculus, thereby providing insight into the operational semantics of functional languages at a broader scope.

By focusing on a minimal core of the Haskell language, the development of MicroHaskell enables controlled experimentation with functional programming concepts and serves as an academic tool for understanding language design and interpreter construction. The report details the language specification, implementation strategy, and key learnings, as well as future work that may be done to extend the language's functionality.

1.1 Haskell

Haskell is a statically typed, purely functional programming language with a strong theoretical foundation in the λ-calculus. It was first standardized in 1990 by a committee of researchers with the goal of consolidating and advancing functional programming research. The language is named after the logician Haskell Brooks Curry, whose work in combinatory logic and functional computation directly influenced its design.

At its core, Haskell supports higher-order functions, immutable data, and first-class functions, consistent with the principles of λ-calculus. Haskell enforces purity, meaning that functions are free of side effects, and all computation is expressed solely through function application. This purity enables referential transparency, wherein expressions can be replaced with their corresponding values without changing program behavior, facilitating formal reasoning and program transformation.

A distinguishing feature of Haskell is its lazy evaluation strategy, also known as non-strict evaluation. Expressions are not evaluated until their results are required, allowing for the construction of infinite data structures and improving modularity by separating the definition of control structures from their execution.

Haskell employs a powerful type system based on Hindley-Milner type inference, extended with advanced features such as algebraic data types, type classes, and polymorphism.

Due to its mathematical foundations and declarative style, Haskell finds use in academic research, formal methods, and select industrial applications where correctness and high-level abstractions are priorities. [1]

1.2 λ-calculus

λ-calculus is a formal system in mathematical logic and computer science for expressing computation based on function abstraction and application. Developed by Alonzo Church in the 1930s, it serves as the theoretical foundation for many functional programming languages such as Haskell.

In λ-calculus, all computation is represented using functions. The system consists of three basic constructs: variables, function abstractions (expressed as $\lambda x.E$, meaning a function with parameter x and body E), and function applications (written as $E_1 E_2$, meaning the application of function E_1 to argument E_2). In the core λ-calculus, there are no primitive data types or control structures; instead, all operations, including arithmetic, conditionals, and data structures, are encoded using functions.

The evaluation of expressions in λ-calculus is governed by a small set of transformation rules, such as alpha-conversion (renaming of bound variables), beta-reduction (applying functions to arguments), and eta-conversion (extensionality). These rules provide the basis for reasoning about program equivalence and transformation.

λ-calculus is Turing complete, meaning it can express any computation that can be performed by a Turing machine. Its emphasis on functions as the sole building blocks of computation directly influences languages like Haskell, which preserve these theoretical properties in practical programming environments.

Haskell's purity, referential transparency, and emphasis on higher-order functions are direct consequences of its foundation in λ-calculus, making the calculus a guiding principle in the language's design. [2]

2 Language Specification

This chapter defines the formal syntax and semantics of the MicroHaskell language. It outlines the supported language constructs, including expressions, functions, bindings, and operators, as well as the rules governing evaluation and scoping.

2.1 Lexical Conventions

MicroHaskell's parser is not sensitive to whitespace. Spaces, tabs, and newlines are used solely to separate tokens, and any extra whitespace beyond this requirement is ignored.

Additionally, MicroHaskell supports only single-line comments, which are initiated with a double minus (`--`). Multi-line or block comments are not supported.

Any character that does not match any of the rules specified below is considered illegal and renders the entire program invalid.

2.1.1 Numeric Literals

The language only provides a single numeric data type; the decimal 32-bit integer. Floating-point numbers and alternative encodings such as hexadecimal are not permitted. An integer literal may include any number of leading zeros, which have no effect on its value. Listing 1 shows the syntax of an integer literal.

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
integer = digit, {digit} ;
```

Listing 1: Grammar of an integer literal.

Examples for valid integer literals include `1`, `42`, and `002517`. Examples for invalid integer literals include `0xCAFE09` and `2.323`.

2.1.2 Identifiers

Identifiers are case-sensitive and may consist of any number of letters (ASCII-only, upper and lower case), digits, underscores, and single quotes (`'`), but they must not start with a digit or a single quote. Listing 2 shows the grammar of an identifier.

```
letter = "A" | "B" | ... | "Z" | "a" | "b" |  ... | "z" ;
identifier ::= ( letter | "_" ), { letter | digit | "_" | "'" } ;
```

Listing 2: Grammar of an identifier.

Examples for valid identifiers include `main`, `adder'`, and `_my_Strange''FUN234`. Examples for invalid identifiers include `'axolotl` and `2517exTinct`.

2.1.3 Keywords

The strings specified in Listing 3 are reserved keywords and cannot be used as variable names or have a value bound to them. Like identifiers, they are case-sensitive.

```
if then else let in infix infixl infixr
```

Listing 3: Reserved keywords.

2.1.4 Operators

Operators may include any number of the characters listed in Listing 4, with the notable restriction that none of the characters allowed in regular identifiers may appear in an operator name. Additionally, while an operator name may include a backslash, a single backslash must not appear as an operator by itself. Furthermore, any regular identifier may be used as an infix operator by enclosing it in back ticks.

```
operator_symbols_without_backslash = "!" | """ | "#" | "$" | "%" | "&" |
"*" | "+" | "." | "/" | "<" | "=" | ">" | "?" | "@" | "^" | "|" | "-" |
"~" | ":" ;
operator_symbols = operator_symbols_without_backslash | "\" ;
operator = symbolic_operator | identifier_operator ;
symbolic_operator = operator_symbols_without_backslash,
{ operator_symbols } ;
identifier_operator = "`", identifier, "`" ;
```

Listing 4: Grammar of an operator.

Examples for valid operators include >>=, <$>, and `` `subtract` ``. Examples for invalid operators include >add< and `;"`. Especially, a custom operator must not conflict with any of the symbols listed in Listing 5, which serve specific syntactic purposes, nor with the double minus (--), which denotes the start of a comment.

```
-> = \ ( ) [ ] , ; `
```

Listing 5: Reserved symbols.

2.2 Syntax

A valid MicroHaskell program is composed of a sequence of top-level function definitions and fixity declarations. Since MicroHaskell's parser is not sensitive to whitespace, semicolons are required to indicate the end of a function definition and fixity declaration. The empty program is also valid. Listing 6 shows the syntax of a valid MicroHaskell program.

```
program = { ( function_definition | fixity_declaration ), ";" } ;
```

Listing 6: Grammar of a MicroHaskell program.

2.2.1 Function Definitions

A function definition introduces a new function and binds it to a name. A function without parameters is effectively equivalent to a constant definition in other programming languages. When defining a function for a symbolic operator, the definition's name must be enclosed in parentheses. Listing 7 shows the grammar of a function definition.

```
function_definition = function_name, { identifier }, "=", expression ;
function_name = identifier | "(", symbolic_operator, ")" ;
```

Listing 7: Grammar of a function definition.

2.2.2 Fixity Declarations

A fixity declaration specifies the associativity and precedence of a function when it is applied as an operator. There are three possible associativities: left (`infixl`), right (`infixr`), and non-associative (`infix`). A fixity declaration starts with the associativity keyword, followed by the precedence level and the function's name. MicroHaskell supports ten precedence levels, ranging from 0 (lowest binding) to 9 (highest binding). When applied to an identifier, the function's name must be enclosed in backticks. Listing 8 shows the grammar of a fixity declaration.

```
fixity_declaration = ( "infixl" | "infixr" | "infix" ), digit,
operator ;
```

Listing 8: Grammar of a fixity declaration.

2.2.3 Expressions

An expression is a syntactic construct that denotes a value and can be evaluated. The default precedences of MicroHaskell's built-in operators are specified in Section 2.3.7. Listing 9 shows the grammar of an expression.

```
expression = let_binding | binary_expression | factor ;
binary_expression = expression, operator, expression ;
factor = "(", expression, ")" | list_literal | if_expression |
anonymous_function | identifier | integer ;
```

Listing 9: Grammar of an expression.

2.2.4 If Expressions

An `if` expression provides conditional branching by selecting between two alternative expressions based on a Boolean condition. Listing 10 shows the grammar of an if expression.

```
if_expression = "if", expression, "then", expression, "else",
expression ;
```

Listing 10: Grammar of an `if` expression.

2.2.5 Function Applications

A function application applies an arbitrary amount of arguments to a given function. Listing 11 shows the grammar of a function application.

```
function_application = factor, { integer | indentifier | list_literal |
"(", expression, ")" } ;
```

Listing 11: Grammar of a function application.

2.2.6 Let Bindings

A `let` binding introduces local definitions of variables or functions within the scope of an expression. It provides a mechanism for naming intermediate results, enabling modularity, readability, and referential sharing. Unlike top-level declarations, `let` is an expression form and can appear wherever an expression is valid. A single `let` expression may contain multiple bindings, separated by semicolons. Additionally, it may contain fixity declarations valid within the created scope. Listing 12 shows the grammar of a `let` binding.

```
let_binding = "let", ( function_definition | fixity_declaration ),
{ ";", ( function_definition | fixity_declaration ) }, "in",
expression ;
```

Listing 12: Grammar of a `let` binding.

2.2.7 Anonymous Functions

An anonymous function is a function definition that is not bound to an identifier. It is defined inline, typically for immediate use, and serves as a means of expressing functional behavior without introducing a named binding. In contrast to regular function definitions, anonymous functions are required to declare at least one parameter. Listing 13 shows the grammar of an anonymous function.

```
anonymous_function = "\", identifier, { identifier }, "->",
expression ;
```

Listing 13: Grammar of an anonymous function.

2.2.8 List Literals

A list literal denotes a finite, ordered sequence of elements enclosed in a syntactic construct. It provides a direct way to construct list values without relying on explicit function applications or constructors. Listing 14 shows the grammar of a list literal.

```
list__literal = "[", expression, { ",", expression }, "]" ;
```

Listing 14: Grammar of a list literal.

2.3 Semantics

This chapter defines the semantics of MicroHaskell, specifying how programs are interpreted and executed. It outlines the evaluation strategy, scoping rules, operator behavior, and the representation of fundamental constructs such as booleans, lists, and functions.

2.3.1 Evaluation Strategy

MicroHaskell empolys lazy evaluation which is a strategy for evaluating expressions where computations are delayed until their results are actually needed. Instead of evaluating everything immediately (eager/strict evaluation), work is deferred and only performed it when required by the program's flow. Additionally, the language ensures referential integrity which allows results of expressions that are needed multiple times to be cached (memoized). This pattern may also be referred to as *call-by-need.* [1]

2.3.2 Untyped Evaluation

MicroHaskell does not employ any form of type analysis. The language supports a single primitive data type, the 32-bit integer, which is used to represent all numeric values. Applying a function to an unsupported argument, such as a value that has not been fully evaluated, may lead to an invalid runtime state.

2.3.3 Integer Values

For performance reasons, integers may be implemented natively in an interpreter.

2.3.4 Boolean Values

MicroHaskell does not define a separate boolean type. Instead, the value `1` is treated as `true`, while all other values are considered `false`.

2.3.5 List Encoding

MicroHaskell does not provide a primitive list type. Instead, lists are represented via Church encoding.[1] The standard library defined in Section 2.4 introduces the functions `cons` and `nil`, which serve as the fundamental constructors for lists. Furthermore, the list literals described in Section 2.2.8 constitute purely syntactic sugar and are desugared into applications of these functions.

[1]https://jnkr.tech/blog/church-scott-encodings-of-adts

2.3.6 Scoping

MicroHaskell empolys lexical scoping: the scope of a variable binding is determined by the static, textual structure of the program. A variable occurrence refers to the nearest enclosing binding of the same name in the source code.

Bindings are introduced in three ways:

1. Function definitions, e.g. `f x = e`, which bind `f` at the top level and each parameter (`x`) within the body `e`.
2. Lambda abstractions, e.g. `\x -> e`, which bind the variable `x` in the expression `e`.
3. Let expressions, e.g. `let x = e1 in e2`, which bind `x` in `e2`. All let bindings are recursive, so the right-hand side may also reference `x`.

Shadowing is permitted: a new binding of a variable name overrides any outer binding of the same name within its scope.

Since MicroHaskell does not perform type analysis, no distinction is made between variable namespaces (functions, operators, or values). All identifiers inhabit a single global namespace, and scoping rules are applied uniformly. In particular, MicroHaskell permits implicit forward declarations, allowing identifiers to be referenced before their defining equations are given. This mechanism enables the definition of mutually recursive functions, as illustrated in Listing 15.

```
-- f0 calls f1
f0 n = if n == 0 then 0 else f1 (n - 1);

-- f1 calls f2
f1 n = if n == 0 then 1 else f2 (n - 1);

-- f2 calls f0
f2 n = if n == 0 then 2 else f0 (n - 1);

main = f0 4;
```

Listing 15: Definition of mutually recursive functions.

2.3.7 Operator Evaluation

MicroHaskell provides a set of built-in operators for basic arithmetic. Each operator has a predefined fixity, specifying its precedence and associativity. For efficiency, these operators may be implemented natively in an interpreter rather than as part of the standard library described in Section 2.4. Built-in operators must behave like regular functions. Their fixity and implementation may be redefined at runtime.

Operator	**Associativity**	**Precedence**	**Description**
`+`	Left	6	Arithmetic addition
`-`	Left	6	Arithmetic subtraction
`*`	Left	7	Arithmetic multiplication
`div`	Left	7	Arithmetic division (integer)
`==`	None	4	Equality
`/=`	None	4	Inequality
`<`	None	4	Less than
`<=`	None	4	Less than or equal
`>`	None	4	Greater than
`>=`	None	4	Greater than or equal

Table 1: Fixities of built-in operators.

2.3.8 Currying

Currying is the process by which functions of multiple arguments are represented as a sequence of functions, each accepting a single argument. [3] In MicroHaskell, all functions are unary at the semantic level. A function that appears to take multiple parameters is therefore desugared into a chain of nested lambda abstractions as shown in Listing 16 and Listing 17.

```
f x y = e
```

Listing 16: Function with two parameters.

```
f = \x -> (\y -> e)
```

Listing 17: Desugared version of the function shown in Listing 16.

This transformation ensures that function application is uniformly left-associative: an expression `f a b` is interpreted as `(f a) b`. Consequently, partial application is naturally supported: providing fewer arguments than the function expects yields another function awaiting the remaining arguments.

2.3.9 Evaluation Entry Point

Every MicroHaskell program is required to define a `main` function, which serves as the entry point of evaluation. When the program is run, `main` is evaluated automatically, and its resulting value is printed to standard output.

2.4 Standard Library

This chapter specifies the standard library of MicroHaskell. The library consists of predefined functions and operators that extend the core language with commonly used functionality and entirely reside within the prelude. These definitions are expressed entirely within the language itself, except where noted for efficiency in Section 2.3.7, and are intended to serve as the canonical basis for program construction. The standard library includes fundamental operations on integers, function combinators, Church-encoded list constructors, and higher-order functions for list processing. Its specification is normative: implementations are required to provide each binding with the behavior described herein, either directly through native implementation or by faithfully reproducing the given definitions in MicroHaskell itself. The entire standard library is part of the prelude and must not be imported.

2.4.1 Arithmetic and Numeric Operations

Table 2 lists functions and operators for performing basic integer arithmetic and aggregate computations on lists of integers

Name	Parameters	Fixity	Description
`abs`	`x`	Default	Computes the absolute value of an integer `x`.
`mod`	`a m`	Left, 7	Computes the modulus of two integers `a` and `b`.
`negate`	`x`	Default	Produces the numeric negation of an integer `x`.
`sum`	`l`	Default	Computes the sum of all elements in a list `l`.
`product`	`l`	Default	Computes the product of all elements in a list `l`.
`maximum`	`l`	Default	Selects the largest element of a list `l`.
`minimum`	`l`	Default	Selects the smallest element of a list `l`.

Table 2: Arithmetic and numeric operations.

2.4.2 Boolean Operations

Table 3 lists definitions for boolean logic expressed through integers as described in Section 2.3.4, including logical connectives and predicates over lists.

Name	Parameters	Fixity	Description
`not`	`n`	Default	Boolean negation (1 $\longrightarrow$ 0, nonzero $\longrightarrow$ 0).
`&&`	`a b`	Right, 3	Boolean AND with short-circuit evaluation.
`||`	`a b`	Right, 3	Boolean OR with short-circuit evaluation.
`any`	`xs p`	Default	Evaluates to `1` if any element in the list `xs` satisfies the predicate `p`, `0` otherwise.
`all`	`xs p`	Default	Evaluates to `1` if all elements in the list `xs` satisfie the predicate `p`, `0` otherwise.

Table 3: Boolean operations.

2.4.3 Function Combinators

Table 4 lists general-purpose higher-order functions and operators that manipulate or combine other functions.

Name	Parameters	Fixity	Description
`flip`	`f x y`	Default	Swaps the arguments `x`, `y` of a binary function `f` (`f x y` → `f y x`).
`id`	`x`	Default	Identity function; evaluates its input `x` unchanged.
`$`	`f x`	Right, 0	Applies function `f` to argument `x`. Used to reduce nesting depth.
`.`	`f g`	Right, 9	Function composition operator for `f` and `g`.

Table 4: Function combinators.

2.4.4 List Construction and Encoding

Table 5 lists primitives for constructing lists using Church encoding, including the empty list, list extension, and list replication.

Name	Parameters	Fixity	Description
`nil`	`consCase nilCase`	Default	Church-encoded empty list.
`cons`	`x xs`	Default	Prepends an element `x` to a Church-encoded list `xs`.
`replicate`	`n x`	Default	Builds a list with `n` copies of the given element `x`.

Table 5: List construction and encoding.

2.4.5 List Membership and Predicates

Table 6 lists functions that inspect properties of lists, such as emptiness or membership of elements.

Name	Parameters	Fixity	Description
`null`	`xs`	Default	Evaluates to `1` if the list `xs` is empty, `0` otherwise.
`elem`	`x xs`	Right, 4	Evaluates to `1` if the element `x` is contained in the list `xs`, `0` otherwise.
`notElem`	`x xs`	Right, 4	Negated version of `elem`.

Table 6: List membership and predicates.

2.4.6 List Processing

Table 7 lists higher-order functions that consume or transform lists, such as folds, mapping, filtering, and structural operations.

Name	Parameters	Fixity	Description
`foldr`	`f z xs`	Default	Right-associative fold over a list.
`foldr`	`f z xs`	Default	Left-associative fold over a list.
`map`	`f xs`	Default	Applies the function `f` to each element of the list `x`.
`filter`	`p xs`	Default	Selects the elements of the list `xs` that satisfy the predicate `p`.
`take`	`n xs`	Default	Extracts the first `n` elements of the list `xs`.
`tail`	`xs`	Default	Produces the list `xs` without its first element.
`head`	`xs`	Default	Extracts the first element of the list `xs`.
`concat`	`xss`	Default	Flattens the list of lists `xss` into a single list.
`length`	`xs`	Default	Computes the length of the list `xs`.
`reverse`	`xs`	Default	Produces the reversed version of the list `xs`.

Table 7: List processing.

2.4.7 List and Sequence Operators

Table 8 lists infix operators for combining lists or indexing into them.

Name	Parameters	Fixity	Description
`++`	`xs ys`	Right, 5	Concatenates the two lists `xs` and `ys`.
`!!`	`xs n`	Left, 9	Extracts the element at the given index `n` from the list `xs`.

Table 8: List and sequence operators.

3 Limitations

Although MicroHaskell is designed to distill many core ideas of lazy functional programming into a compact and analyzable form, its minimal design imposes a number of important limitations. These restrictions are partly deliberate, but they also highlight areas where potential further development could be made.

3.0.1 Inefficiency of Church Encodings

Lists in MicroHaskell are expressed via Church encoding, which represents structures as higher-order functions rather than as native data types. While this has theoretical elegance and allows all data to be represented within the λ-calculus, it comes at a cost:

- Many operations, such as `length`, `reverse`, or `concat`, require traversing the entire structure even to obtain partial information.
- Pattern matching is not natively supported, which complicates reasoning and often prevents efficient short-circuiting.
- The encoding hides structural sharing, so memory efficiency and incremental evaluation (important for laziness) are less predictable.

In practice, it can be assumed that Church-encoded data structures are significantly less efficient than native algebraic data types.

3.0.2 Invalid Programs and Debugging Difficulties

MicroHaskell does not include a type system or static analysis. As a result, it is possible to write programs that correspond to ill-formed lambda terms at runtime. For instance:

- A function may be applied to too few arguments, leaving a partially applied function where an integer is expected.
- Built-in operators may be applied to non-numeric values.

Such failures typically manifest as invalid reductions in the λ-calculus representation. From the perspective of a programmer, this makes debugging difficult: the exact point of failure is often non-obvious, and runtime error messages are limited.

3.0.3 Restricted Data Model

While 32-bit integers are sufficient to encode booleans and Church lists of integers, this restricts expressiveness:

- No distinction is made between integers, booleans, or function values, which can obscure program intent.
- Arithmetic is limited to 32-bit integer operations, excluding floating-point or arbitrary-precision numbers.
- The absence of algebraic data types prevents the natural definition of domain-specific structures.

3.0.4 No Concurrency or Side Effects

MicroHaskell is purely functional and does not provide primitives for I/O, concurrency, or interaction with the external environment beyond the evaluation and printing of `main`. This makes it unsuitable for real-world applications and constrains it to being a closed computational model.

4 Implementation Specifics

This chapter describes specific aspects of the reference implementation of MicroHaskell. While the preceding chapters establish the abstract syntax and formal semantics of the language, the reference implementation provides a concrete realization of these concepts. The purpose of this chapter is not to redefine the language itself, but to document certain implementation choices that were made.

4.1 AST Rewriting

Since MicroHaskell permits the definition of custom infix operators together with their associativity and precedence, expressions involving such operators cannot be parsed directly into an AST in a single pass, as the relevant fixity information is not yet available. Instead, these expressions are initially parsed into a flat sequence containing both operators and subexpressions. During the subsequent name analysis phase, fixity declarations are collected and stored in the appropriate symbol tables. Once this information has been established, the AST is traversed and each flat expression is restructured according to the fixities of the operators it contains.

For this purpose, an operator precedence parser is employed. An operator precedence parser is a parsing technique designed to handle expressions involving infix operators with varying precedences and associativities. The central idea is to assign each operator a numerical precedence level and an associativity (left, right, or non-associative). Parsing proceeds by comparing the precedence of the operator most recently seen with that of the next operator to determine whether to shift (read further input) or reduce (form a subexpression). [4]

This mechanism ensures that higher-precedence operators bind more tightly than lower-precedence ones, and that associativity resolves ambiguity among operators of equal precedence. The resulting AST is consistent with the declared precedence and associativity rules.

Figure 1 illustrates the process in which a definition with a body containing an artiehmetic expression is parsed and rewritten.

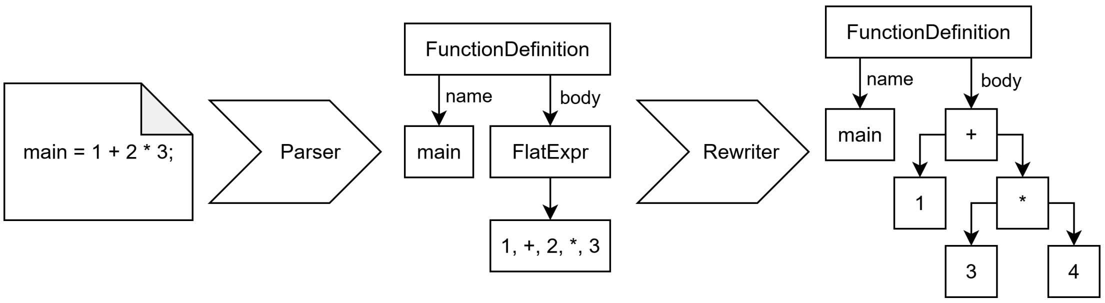

Figure 1: The parsing process of an expression.

4.2 Intermediate Representation

After parsing a program into an AST, the interpreter translates it into an intermediate representation (IR) that closely follows the constructs of the untyped λ-calculus established in Section 1.2. At its core, the IR is built around the same three fundamental concepts of lambda (function) abstractions, applications, and variables. For efficiency, the interpreter does not recursively apply the transformation rules described, but instead evaluates the IR iteratively. This design requires the representation of a closure, which pairs a lambda abstraction with the environment that captures its bound variables. In addition, the IR includes dedicated structures for representing native integers and for functions that are implemented directly within the interpreter.

4.3 Recursion

As discussed Section 2.3.6, all variable bindings introduced by a `let` expression or by a sequence of top-level declarations are recursive by default. This means that within their scope, each binding may refer not only to itself (direct recursion) but also to the other bindings defined in the same group (mutual recursion). Listing 18 shows a program that employs both direct as well as mutual recursion to determine whether the 5th Fibonacci number is even.

```
-- Directly recursive function `fib`.
fib n =
    if n == 0 then 0
    else if n == 1 then 1
    else fib (n - 1) + fib (n - 2);

-- Mutually recursive function `even` and `odd`.
even n = if n == 0 then 1 else odd (n - 1);
odd n = if n == 0 then 0 else even (n - 1);

main = even $ fib 5;
```

Listing 18: Example program exhibiting both direct and mutual recursion.

4.3.1 Direct Recursion

The untyped λ-calculus does not provide recursion as a primitive construct. Instead, recursion can be expressed by means of a fixed-point operator. [2] The canonical example is the Y combinator, defined as

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

For any functional F, the term YF satisfies the fixed-point equation

$$YF \twoheadrightarrow F(YF).$$

This property allows the definition of directly recursive functions. For example, the Fibonacci function as defined in Listing 18 can be expressed in λ-calculus by first defining a higher-order functional

$$\begin{gathered} F = \lambda f.\lambda n.\ \text{if } n = 0 \text{ then } 0 \\ \text{else if } n = 1 \text{ then } 1 \\ \text{else } f(n-1) + f(n-2), \end{gathered}$$

and then setting

$$\text{fibonacci} = YF.$$

Here, the recursive self-reference of *fibonacci* is mediated entirely by the fixed-point operator: Y supplies the function f with its own definition, thereby closing the recursion.

4.3.2 Mutual Recursion

Mutual recursion arises when several functions are defined in terms of one another. For example, the pair of functions for determining whether a given number is odd or even as described in Listing 18

$$\begin{gathered} \text{even}(n) = \text{if } n = 0 \text{ then True else odd}(n-1), \\ \text{odd}(n) = \text{if } n = 0 \text{ then False else even}(n-1), \end{gathered}$$

are interdependent: each definition references the other. Formally, mutual recursion requires solving a system of equations of the form

$$f = F(f, g), g = F(f, g),$$

that is, computing the simultaneous fixed point of a tuple of functionals. The standard Y combinator computes a fixed point for a single functional, yielding solutions of the form

$$f = YF = F(f)$$

It does not extend directly to the case of multiple mutually recursive definitions, since no mechanism is provided to construct a tuple of functions and solve for their joint fixed point simultaneously. In the λ-calculus, one can in principle define generalized fixed-point combinators, such as Y_2, which produce pairs (f, g) satisfying

$$f = F(f, g), g = F(f, g),$$

but such constructions require additional machinery, such as tuple encodings, and are considerably more involved. Instead, a mechanism may be employed that automatically rewrites mutual recursion as direct recursion which can be expressed using the canonical Y combinator.

4.3.3 Tarjan's Strongly Connected Components Algorithm

A strongly connected component (SCC) of a directed graph is a maximal subset of vertices in which every vertex is reachable from every other.

Tarjan's algorithm is a depth-first search (DFS)–based method for computing all SCCs of a directed graph in a single traversal. It assigns each vertex a unique discovery index as it is first visited and maintains, for each vertex, a *low-link* value indicating the smallest discovery index reachable from that vertex (including itself). A stack records the current active path of the DFS. When the algorithm identifies a vertex whose discovery index equals its low-link value, it recognizes the root of an SCC; all vertices on the stack up to that root form one strongly connected component.

The algorithm runs in linear time with respect to the number of vertices and edges, i.e. $O(V + E)$, making it highly efficient for large graphs. [5]

4.3.4 Identifying Recursive Applicative Relationships

For every MicroHaskell program, one can construct a corresponding directed graph in which each function f is represented by a vertex. Whenever the body of f contains an application of another function g, a directed edge is added from f to g. Notably, it may also apply that $f = g$; in this case, a self-edge will be created.

Applying this method to the program shown in Listing 18 yields the graph shown in Figure 2.

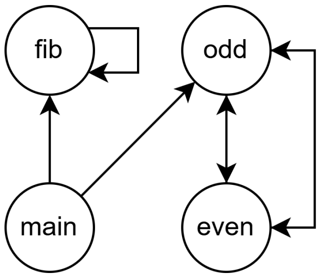

Figure 2: Application graph constructed for Listing 18.

By applying Tarjan's SCC algorithm to the graph shwon in Figure 2, the following set of strongly connected components are determined:

$$\{\{\text{main}\}, \{\text{fib}\}, \{\text{odd}, \text{even}\}\}$$

If an SCC contains more than one vertex, all functions within that component are mutually recursive. If the SCC consists of a single vertex f, two cases must be distinguished: either f is directly recursive, or it is non-recursive. This distinction can be made by checking whether the graph includes a self-edge from f to itself.

4.3.5 Applying Recursive Functions

During program analysis, each function is represented by an entry in a symbol table. Alongside its definition, the function is annotated with its recursion property (classified as non-recursive, directly recursive, or mutually recursive). Furthermore, every function is assigned a unique integer identifier, hereafter referred to as a *tag*.

When generating the intermediate representation for a function application, the function is first retrieved from the symbol table.

If it is non-recursive, its body can be emitted immediatly without further processing.

If it is directly recursive, the function body is wrapped with the Y combinator, as described in Section 4.3.1, thereby enabling recursion immediately.

In the case of mutual recursion, a dispatcher pattern is employed. Specifically, a single enclosing function `mutual_dispatch` is generated, which maps tags to their corresponding function bodies. A call to a mutually recursive function is translated into an application of this dispatcher with the function's tag and the arguments passed in the source application. The dispatcher then performs a case distinction to select and evaluate the appropriate function body. To facilitate recursive calls, the dispatcher is itself wrapped in the Y combinator.

Listing 19 illustrates how the program from Listing 18 is transformed to account for the mutually recursive relationship between even (tag $= 0$) and odd (tag $= 1$) at a high level. This rewritten version now only contains direct recursion which is natively supported through the Y combinator.

```
-- Directly recursive function `fib`.
fib n =
    if n == 0 then 0
    else if n == 1 then 1
    else fib (n - 1) + fib (n - 2);

-- Unified recursive function
dispatch tag =
  if tag == 0 then \n -> if n == 0 then 1 else dispatch 1 (n - 1)
  else \n -> if n == 0 then 0 else dispatch 0 (n - 1);

-- Named wrappers for each function
even = dispatch 0;
odd = dispatch 1;

main = even $ fib 5;
```

Listing 19: Rewritten version of program shown in Listing 18

4.4 Deferred Computations

As stated in Section 2.3.1, MicroHaskell's evaluation follows call-by-need semantics: expressions are not evaluated immediately, but only when their results are required. This deferred evaluation is implemented using *thunks*.

Conceptually, a thunk can be understood as a suspended computation which is essentially a lambda abstraction with no arguments that, when invoked, evaluates the stored expression. For example, instead of binding a variable x directly to an expression e, the runtime binds x to a thunk $\lambda . \to e$. Accessing x then amounts to applying this trivial function, which, in turn, triggers the evaluation of e.

Once a thunk has been evaluated, its result is cached (memoized) and the thunk is essentially replaced by the computed value. This ensures that every expression is evaluated at most once, in contrast to a pure call-by-name strategy where recomputation may occur. The concept of a deferred compuation is necessary to express recursion in the lambda-calculus-based interpreter utilized by the language. The thunk only exists during runtime and is therefore not strictly part of MicroHaskell's intermediate representation.

4.5 Conditional Evaluation

Due to MicroHaskell employing lazy evaluation, conditional branching is realized as a built-in function `if`. This function expects three arguments: the condition, the then-branch, and the else-branch. It first evaluates the condition and, depending on the result, evaluates exactly one of the two branches accordingly.

4.6 REPL

When the MicroHaskell interpreter is started without arguments, it launches the REPL (**R**ead-**E**valuate-**P**rint-**L**oop), which reads expressions entered by the user, evaluates them, and prints their results in an infinite loop. The REPL maintains a persistent execution context that is incrementally extended with new definitions provided during the session. Input is first processed by the lexer, which tokenizes the source code; these tokens are also used to produce a syntax-highlighted rendering of the entered definition. Within the REPL, the only function that may be redefined is the program entry point `main`, which is also the sole function evaluated immediately after being entered.

5 Application Examples

This chapter presents a selection of example programs written in MicroHaskell. The purpose of these examples is to illustrate how the language's features can be combined to express non-trivial computations within its minimalist core.

5.0.1 Computing Prime Numbers

Listing 20 defines helper functions to test divisibility and primality, then generates all prime numbers up to `30`, producing `[2,3,5,7,11,13,17,19,23,29]`.

```
upto lo hi =
  if lo <= hi
    then [lo] ++ (upto (lo + 1) hi)
    else [];

divides d n = (n `mod` d) == 0;

isPrime n =
  if n <= 1 then 0
  else not $ any (\d -> divides d n) (upto 2 (n - 1));

primesUpTo n = filter isPrime (upto 2 n);

main = primesUpTo 30;    -- [2,3,5,7,11,13,17,19,23,29]
```

Listing 20: Collects all primes in the range from 0 to 30.

5.0.2 Insertion Sort

Listing 21 implements insertion sort by repeatedly inserting elements into their correct position using `filter`, and sorts the list `[5, 6, 2, 1, 1]` into `[1, 1, 2, 5, 6]`.

```
insert x xs = filter (\y -> y < x) xs ++ [x] ++ filter (\y -> y >= x)
xs;

insertionSort xs = foldr insert [] xs;

main = insertionSort [5, 6, 2, 1, 1];   --- [1, 1, 2, 5, 6}
```

Listing 21: Insertion sort.

5.0.3 Function Composition

Listing 22 defines a squaring function and computes the sum of the squares of `[1, 3, 5]`, yielding `35`.

```
square x = x * x;
main = (sum . map square) [1, 3, 5];  -- 35
```

Listing 22: Composition and point-free style over a literal list.

5.0.4 Pipeline Operator

Listing 23 defines a pipeline operator and uses it to build the numbers from `1` to `8`, square them, and sum the results, producing `204`.

```
infixl 1 |>;
(|>) x f = f x;

upto lo hi =
  if lo <= hi
    then [lo] ++ (upto (lo + 1) hi)
    else [];

main =
  8
  |> (\n -> upto 1 n)
  |> (map (\x -> x * x))
  |> sum;                 -- 204
```

Listing 23: Custom pipeline operator.

5.0.5 Infinite List Operation

Listing 24 defines an infinite list of natural numbers, maps it to their squares, and lazily evaluates only the 10th square, yielding `100`.

```
natsFrom n = cons n (natsFrom (n + 1));

squares = map (\x -> x * x) (natsFrom 1);

main = squares !! 9;  -- 100 (the 10th square)
```

Listing 24: Infinite list operation.

6 Outlook and Future Work

The current design of MicroHaskell captures a minimal fragment of Haskell, centered around lazy evaluation and higher-order functions. While this suffices as an academic and experimental language, several directions remain for future extension:

1. **Algebraic data types and pattern matching.** Native support for algebraic data types would replace the reliance on Church encodings and enable more direct expression of common recursive structures. Pattern matching would complement this, providing a canonical mechanism for case analysis.
2. **Static type system.** A static layer, potentially based on Hindley-Milner inference, could improve program safety and enable richer program analysis. In its current state, error reporting is minimal and debugging larger programs difficult.
3. **Module and namespace system.** Introducing modules would support larger-scale program organization and facilitate code reuse, an important step toward scaling beyond toy programs. This would include support for a more fully-featured standard library that may not reside entirely within prelude.
4. **Concurrency and parallelism.** Incorporating lightweight concurrency primitives would open opportunities to study the interaction between non-strict semantics and parallel execution.
5. **Side Effects.** Currently, MicroHaskell provides no mechanism for handling functions that produce side effects, such as interacting with the operating system to create or write to files, communicate over sockets, or access other external resources. A possible extension would be to adopt an approach similar to that used in Haskell.

These extensions would move MicroHaskell closer to the expressivity of full Haskell while retaining its tiny core calculus.

Bibliography

[1] M. Lipovaca, *Learn You a Haskell for Great Good!: A Beginner's Guide.* San Francisco: No Starch Press, 2011.

[2] J. Alama and J. Korbmacher, “The Lambda Calculus,” *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2024. Accessed: Sep. 04, 2025. [Online]. Available: https://plato.stanford.edu/archives/win2024/entries/lambda-calculus/

[3] “Currying - HaskellWiki.” Accessed: Sep. 04, 2025. [Online]. Available: https://wiki.haskell.org/index.php?title=Currying

[4] T. Niemann, “Operator Precedence Parsing.” [Online]. Available: https://www.epaperpress.com/oper/download/OperatorPrecedenceParsing.pdf

[5] R. Tarjan, “Depth-first search and linear graph algorithms,” in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, Oct. 1971, pp. 114–121. doi: 10.1109/SWAT.1971.10.