Marsh: Demonstrating a Strong and Static Type-and-Effect System for Shell Scripting Languages

Soren Mortensen

Bachelor of Science in Computer Science
The University of Bath
2021–2022

Marsh: Demonstrating a Strong and Static Type-and-Effect System for Shell Scripting Languages

Submitted by: Soren Mortensen

Abstract

Shell scripting languages such as Bash are very weakly typed, and in many cases this directly leads to logic errors, difficult-to-read code, and painful refactoring.

We suggest that a more powerful type system could go some way towards alleviating these issues. The Functional Machine Calculus (FMC) represents a possible basis for an implementation of such a type system, even for an imperative language.

We present Marsh, a proof-of-concept implementation of such a shell scripting language, which is translated to FMC terms for execution on a stack machine. The language is strongly and statically typed, and supports function declarations and calls, conditional branching, and both input/output and non-determinism effects. We also introduce a representation of the FMC using numerical indices rather than named variables, in the style of De Bruijn, for efficient evaluation.

Contents

1 Introduction **2**
1.1 Objective . . . 2

2 Literature Survey **3**
2.1 Introduction . . . 3
2.2 Shell Scripting . . . 3
2.2.1 Thompson Shell & PWB Shell . . . 4
2.2.2 Bourne Shell . . . 4
2.2.3 Bash . . . 5
2.2.4 Fish . . . 6
2.2.5 Nushell . . . 6
2.2.6 Ion . . . 7
2.3 Effects . . . 7
2.3.1 Functional Machine Calculus . . . 8

3 Requirements **10**
3.1 High-Level Goals . . . 10
3.2 Detailed Requirements . . . 11

4 Design **13**
4.1 Syntax . . . 13
4.1.1 Expressions & Statements . . . 13
4.1.2 Locations . . . 14
4.1.3 Function Calls . . . 15
4.2 Types . . . 16
4.2.1 Calling Convention . . . 17
4.2.2 Variables . . . 17
4.3 FMC Term Representation . . . 18
4.3.1 De Bruijn Indices . . . 18

5 Implementation **21**
5.1 Overview . . . 21
5.2 Lexing . . . 22
5.3 Parsing . . . 23
5.3.1 Possible Alternatives . . . 24
5.3.2 Abstract Syntax Tree . . . 25
5.4 Type Checking . . . 26

5.5 Translation . . . 27
5.5.1 AST to FMC Terms . . . 27
5.5.2 Named to Indexed Terms . . . 27
5.6 Beta Reduction . . . 28
5.6.1 Named Terms . . . 28
5.6.2 Indexed Terms . . . 28
5.7 Evaluation . . . 28
5.8 Testing . . . 29

6 Conclusions 31
6.1 Results . . . 31
6.2 Future Work . . . 32
6.3 Critique . . . 32

7 Bibliography 33

A Selected Source Code 37
A.1 Grammar . . . 37
A.2 AST . . . 45

List of Figures

4.1 Two Representations of $\lambda x.\lambda y.x$ 19
4.2 Two Representations of $\lambda z.(\lambda x.x)(\lambda y.z)$ 19
4.3 Two Representations of $\langle x\rangle.\langle y\rangle.x$ 20
4.4 Two Representations of $\langle z\rangle.[\langle y\rangle.z].\langle x\rangle.x$ 20

List of Listings

2.1 Comma Separated List of Files (Bash) 6
2.2 Comma Separated List of Files (Ion) 7
4.1 Example Marsh Function (`print_sum`) 13
4.2 Pop Expression Examples 13
4.3 Optional Final Expression 14
4.4 Constant Initialised With a Block 14
4.5 Equivalent Locations 15
4.6 Ambiguous Usage of `read_int` 15
4.7 Required Postfix Parentheses 16
4.8 Exploitation of Partial Application 16
4.9 Call Operator 16
4.10 Interference with the λ Stack 17
4.11 Simple Usage of a Variable 18
4.12 Variable in Function Body 18
5.1 Example Function 22
5.2 Equivalent Token Names 22
5.3 Snippet of Lexer `match` Declaration 23
5.4 Simple Production Rules 23
5.5 `PrimaryExpr` Production 24
5.6 `Literal` AST Type 25
5.7 `Func` AST Type 25
5.8 `Type` AST Type 26
5.9 `Effect` AST Type 26
5.10 `TypeCheck` Trait 26
5.11 `Translate` Trait 27
5.12 `Memory` Trait 29
5.13 Test of Type-Checking an `Expr::Pop` 30
6.1 Example Higher-Order Function 31
6.2 Non-Determinism and Output 31

Acknowledgements

I would like to thank my supervisor, Dr Willem Heijltjes, for his guidance, insight, and patience. Without him, I would not have been able to transform my initial, far too widely scoped idea into the project it is today.

I would also like to thank my wonderful girlfriend Gemma for her proofreading, rubber-ducking and, of course, also patience.

And I would like to thank my parents, for letting me ramble about syntax and enthuse about Rust.

Finally, I would like to thank Teo for the manele recommendations.

Chapter 1

Introduction

Shell scripting languages have existed for more than 50 years [32], and in the time since the release of the Bourne shell in 1979 [24], their fundamental nature and even most of their feature sets have not changed significantly. Their purpose centres around manipulating text and automating repetitive tasks, primarily by invoking other programs, and it is for this reason as well as a myriad of others that their type systems are simplistic and weakly enforced.

The lack of a powerful type system limits the options available to the programmer for how to structure scripts and manipulate data. This, among a number of other idiosyncracies, also makes writing shell scripts an error-prone experience, which has led to the development of static analysis tools such as ShellCheck [36]. However, due to the fundamental design of these languages' type systems, static analysis tools are only capable of finding a subset of possible errors, and mainly rely on pattern matching to do so.

Attempts have been made to address some of these issues, by building more modern shell scripting languages with stronger type systems, more powerful built-in tools, and more intuitive and consistent syntax. None of these, however, are truly strongly-typed languages, and so retain many of the problems and limitations of their predecessors.

1.1 Objective

We will present a proof-of-concept implementation of a shell scripting language making use of the Functional Machine Calculus [2] as the basis for a strong, static type-and-effect system. This implementation will consist of a parser, type-checker and interpreter for a shell scripting language called Marsh. The language will be imperative, and will allow the programmer to express effects—including reading from the standard input stream, writing to the standard output and standard error streams, and generating non-deterministic values—in the types of functions.

Chapter 2

Literature and Technology Survey

2.1 Introduction

In this project, we attempt to bring together two disparate areas of research and innovation: shell scripting, which has to a significant degree evolved through a process of incremental improvement of existing software implementations to meet the day-to-day needs of users; and computational effects, the study of which is formal and grounded in an understanding of type and category theory.

This chapter will explore the history of shell scripting, covering its origins and evolution as well as the historical reasons for the lack of strong type systems in even modern shells. We will argue that shell scripting languages stand to benefit from strong, static typing, but that doing so in a complete fashion is not realistic without a type system that includes a notion of computational effects. We will explore several possible representations of effects, and propose that the Functional Machine Calculus forms a suitable basis for such a type system.

2.2 Shell Scripting

The command-line interface has existed for almost as long as interactive computer systems themselves. Having evolved from interfaces based on teletype (TTY) machines, early command-line interfaces such as the TOPS-10's resident monitor interface were capable of reading commands entered by the user and parsing and expanding their arguments [4, 5]. These commands included not only launching user-provided programs but also direct interaction with the operating system and hardware.

However, it's important to distinguish command-line interfaces in general from shell programs specifically. The term "shell" was coined by Louis Pouzin in 1964 to describe a command language for the Multics operating system [32]. One important difference is that this shell, unlike its predecessors, was not built in to the operating system, but a command "called automatically by the supervisor whenever a user types in some message at [their] console" [33]. This shell program included control flow constructs, such as looping and conditionals, and introduced the notion of substitution, allowing the user to write a script using variable names that would be substituted with values when called [31].

Another important distinction is that this shell program could process not only commands typed at an interactive prompt but also those stored in a file. Together with the inclusion of control flow, this marked the beginning of the evolution of the command-line interface into something akin to a programming language.

2.2.1 Thompson Shell & PWB Shell

It was the Thompson shell, written by Ken Thompson and released with Unix in 1971, that introduced much of the syntax and basic design still present in today's shell scripting languages. Commands consist of a space-separated list of arguments, the first of which is the name of the program to run [25]. They could be chained together by the use of *pipes* (denoted by `|`), which connect the output of one command to the input of the next, and *redirection* of these input and output streams to and from files could be achieved with angle brackets (`>` and `<`) [25].

However, the Thompson shell was of relatively limited utility as a scripting language, having been designed first and foremost for use as an interactive prompt, and was soon adapted by Mashey and several others into the PWB shell, which included several more advanced control flow constructs [25].

Command substitution allowed the user to launch another instance of the shell inside a command pipeline to "[combine] the outputs of several sequentially executed commands into a stream to be processed by a pipeline" [25]. Certain characters were given special meanings such as `?` and `*`, which would cause the shell to expand command arguments into lists of all files that match the provided pattern, replacing `?` with any single character and `*` with any string of characters other than the path separator (`/`). Variable expansion allowed the user to dereference variable names using `$`, inserting their values into strings.

These are highly convenient features for the user of the shell in that, for example, a command `foo /bar/baz/quux1 /bar/baz/quux2 /bar/baz/quux3` could instead be written `foo /bar/baz/quux?`. If additional processing needed to be performed on the output of that command using some command `corge`, it could be tacked on the end with a pipe, as: `foo /bar/baz/quux? | corge`.

It is important to note, though, that these features all rely on the shell's treatment of commands as malleable strings, which makes it difficult to predict exactly what the structure of the final command will be. For example, the replacement of a pattern containing the character `?` with a space-separated list of filename matches turns a single argument into a sequence of an unknown number of arguments. This has the potential to change the meanings of any other arguments appearing after it and produce unexpected behaviour.

2.2.2 Bourne Shell

The Bourne shell—once again named after its creator, Stephen Bourne—was introduced in version 7 of Unix in 1979 [34]. This shell, rather than evolving from a command-line interface into a programming language, was designed from scratch to address the shortcomings of its predecessors. Despite this, it kept the fundamental idea of treating shell code as strings, inside which substitutions could be performed quite freely. As Bourne described it in an interview in 2009 [7]:

> [You] wouldn't want to be typing commands and have all the strings quoted like you would in C, because most things you type are simply uninterpreted strings. You don't want to type `ls directory` and have the directory name in string quotes because that would be such a royal pain. Also, spaces are used to separate arguments to commands. The basic design is driven from there and that determines how you represent strings in the language, which is as un-interpreted text.

The Bourne shell was successful, even to the degree that it is still present on almost all modern Unix-like systems. This success was instrumental in establishing the shell as both an interactive and a scripting language, and one central to the Unix model of user interaction. Later shells such as Bash were based in part on the Bourne shell, as its feature set was deemed essential for any shell scripting language.

2.2.3 Bash

Bash, or the Bourne Again Shell, was released in 1989 as a replacement for the Bourne shell, and has since then become perhaps the most commonly used shell program—Bash is the default login shell for users of most distributions of Linux [8] (and, until recently, macOS [1]). It retains almost all features of the Bourne shell and adds many more, such as command line history, functions, and support for integers and lists [34].

Data that comes from an external source (the output of a command, the contents of a file, etc.) is generally interpreted by Bash as either a string or a list of strings. The task of manipulating this data is outsourced to external programs such as `awk`, `sed`, and `jq`, which must operate on input about which nothing more is known than that it is text. This means that each program must individually attempt to parse its input into the structure it expects; the source code of `jq` contains a JSON parser, for example [19].

Information is input to a Bash script in a variety of ways—it may parse arguments passed to it, call commands and read from their standard output, or examine the exit codes of other commands—and likewise it can output information in various ways. This wide variety of ways to perform essentially the same task is pervasive in Bash, and can make understanding how to use a script difficult, as well as obscuring the flow of data through the program.

Listing 2.1 is a Bash script that produces a comma-separated list of the files in the current directory. First it runs the external `ls` command, which prints a space-separated list of files, and stores its output in the variable `files`. Then the script loops through each space-separated value in `files`, setting `out` to the concatenation of `${out:+$out, }` (which is an expression that expands to the value of `out` followed by a comma and a space only if `out` is non-empty) and `file`. At the end, it prints the value of `out`, which at that point contains a comma-separated list of files.

This is a naïve implementation, but one that a beginner (or even intermediate) user of Bash could be reasonably expected to write. On the surface it looks sensible, and the ShellCheck static analysis tool [36] finds no issues with it. When run in a

```bash
#!/usr/bin/env bash
files=$(ls)
out=""

for file in $files
do
    out=${out:+$out, }$file
done

echo "$out"
```

Listing 2.1: Comma Separated List of Files (Bash)

directory containing itself and two files `foo` and `bar`, this script produces the output "`bar, bash-list.sh, foo`", as expected.

Unfortunately, if a file is added in that directory named `this filename contains spaces`, it outputs "`bar, bash-list.sh, foo, this, filename, contains, spaces`", which is incorrect. The reason for this is that, while Bash does have a list data type, this script does not make use of it. `ls` prints the list of files separated by whitespace, and that output is simply stored in `files` as a string. The loop on line 5 loops through each space-separated substring of `files`, so splits the filename `this filename contains spaces` into four separate elements.

It is possible to rewrite this script to avoid this issue, along with a number of other issues that arise when trying to parse the output of `ls` [40], but doing so is not straightforward, and might require the use of other external commands such as `find` or `xargs`. This is of course only one example, but serves to illustrate some of the problems associated with Bash; if it were possible to express the type of `ls` so that it is understood as returning a list of filenames, for example, this problem would not arise.

2.2.4 Fish

The Friendly Interactive Shell (`fish`) focuses on "usability and interactive use" [10], attempting to eliminate many of the idiosyncracies of older shells. Its syntax is more consistent, and makes heavier use of keywords rather than symbols.

Unfortunately, although the particular example in Listing 2.1 can be rewritten in Fish to avoid splitting filenames containing spaces using the `string` builtin command, many fundamental problems remain, since the differences are largely superficial or incremental. In particular, it is still possible to write Fish scripts that have similar string-splitting problems, because Fish in no way forces the user to use its more modern features to avoid such issues. Like Bash, Fish suffers from a lack of information about the type of the data output by external commands like `ls`.

2.2.5 Nushell

Nushell (`nu`) [29] is a project whose stated goal is to "take the Unix philosophy of shells, where pipes connect simple commands together, and bring it to the modern style of development" [28]. Nu focuses on understanding the structure of the data it is processing, returning even the list of files in the current directory as a structured table.

Data is modelled as either “simple” or “structured”; structured data, such as rows, lists, tables, and blocks, can be manipulated using built-in commands [28]. Custom commands can also be written that are capable of acting on structured data, by annotating their inputs with types.

Nu’s primary method for composing multiple commands is through pipelines, where the output of one command is connected to the input of the next command. In this way, Nu eliminates a lot of the inconsistency arising from the multiple available methods to transfer data between commands in shells like Bash and Fish.

However, type annotations for custom command parameters are optional (when left off, types default to `any`), and all type-checking happens at runtime. In addition, Nu’s type system is not sufficiently general; while a type `block` exists to represent a block of code, this is an erased type with no information about the parameter or return types of the block. This makes it difficult or impossible to define a custom command implementing the `map` operation, for example, and makes the programmer reliant on built-in implementations of various methods of structured data manipulation.

2.2.6 Ion

Ion (`ion`) is a shell developed primarily for RedoxOS [17], and focuses on providing a simple and powerful syntax, as well as security and performance, citing specifically the ShellShock bug [6] as a motivating factor.

Ion supports type annotations for variables and function parameters, but its list of supported types is short, consisting only of `str`, `bool`, `int`, `float`, and arrays and key-value maps of those types [18]. In this way, its type system is even more limited than Nu’s, and as a result, it suffers from similar problems to Bash.

```
#!/usr/bin/env ion
let files = [ @(ls) ]
echo $join(files ", ")
```

Listing 2.2: Comma Separated List of Files (Ion)

Listing 2.2 is an Ion implementation of the same operation as Listing 2.1. The `ls` command is run, and its output split on whitespace into an array. Then the elements of the array are joined by the separator string `", "`, and the resulting string is printed. This implementation has the same whitespace-splitting issue as the one in Listing 2.1. Ion once again suffers from a lack of information about the type of the data output by `ls`.

2.3 Effects

The problem of controlling, representing, tracking and/or analysing *computational effects*—in particular, *side effects*—in programming languages has been the subject of much study. Ghezzi defines a side effect as “the modification of the nonlocal environment" [13]. For example, this might be by altering the value of a global variable, making a modification through a parameter passed by reference, or performing I/O [37]; generally, modification of state.

A major motivation of the academic focus on this topic is the degree to which effects interfere with the ability of a programmer or automated tool to understand the complete consequences of evaluating a section of code, and therefore reduce the ease of modification of any code that interfaces with it. From a more formal point of view, effects restrict the degree to which equational reasoning can be used to perform transformations on programs [30].

Purely functional programming languages such as Haskell take the approach of separating computations that can produce side effects from those that cannot, such as through the use of monads. This is a necessary condition for the property of referential transparency, which is foundational to functional programming: any call to a function can be replaced with the value it evaluates to, with no effect on the semantics of the overall program.

However, the approach of restricting or separating effects in this way is often too strict in practice, particularly in the context of shell scripting. Programs written in shell scripting languages are extremely effect-dense, because one of their primary purposes is to provide an interface to perform tasks concerned with manipulating state. It is our view, therefore, that an implementation of a type system for such a language must include a notion of effects. Excluding this risks discarding essential information about the behaviour of programs, and would not offer a significant advantage over the state of the art.

For example, consider a function written in a shell scripting language whose input is solely provided by the user entering text at the command line. This function could not be given a useful type by a system that did not include a notion of effects, because it does not take any parameters.

A number of approaches have been proposed for the representation of computational effects, including those based on monads [26, 30] and thunks [16, 21] in purely functional languages. More recently, "type-and-effect systems" [15, 22, 23, 35] have gained traction as a method of guaranteeing deterministic semantics [3] and verifying atomicity [11] in parallel programming; guarding against privileged operations [12]; and expressing uniqueness and immutability [14].

2.3.1 Functional Machine Calculus

Peyton Jones and Wadler described a system [30] for representing input/output effects in purely functional languages based on monads [26, 39, 38]. This approach introduces a type `IO` that represents an action that may perform an input/output operation when evaluated, but is in the meantime simply a value that can be passed around as usual [30]. Input/output actions can be composed using monad transformers, allowing for the sequencing of actions.

However, this composition using monad transformers quickly becomes tiresome. In particular, it is difficult to compose monads that represent the manipulation of more than one different kind of state, such as I/O and array operations [30]. In practice, as Barrett, Heijltjes and McCusker point out, "multiple effects are combined by building a stack of monad transformers, which can become unwieldy for the programmer" [2].

As an attempt to address this criticism, rather than extending the λ-calculus itself to represent effects, Barrett, Heijltjes and McCusker describe the Functional Machine Calculus, a model of higher-order computation whose design fundamentally includes a notion of effects [2]. As a result of the combination of *locations* and *sequencing*, two generalisations of the λ-calculus whose combination is unique to the FMC, imperative programs may be type-checked while taking their computational effects into account and evaluated on a stack machine.

It is our view that a strongly typed shell scripting language would benefit from the use of the FMC as a basis for its type-and-effect system, as its locations allow for an easy representation of traditional shell scripting concepts such as input/output streams and pipelines, and its support for sequences of terms maps well onto the traditionally imperative style of shell scripting languages.

Chapter 3

Requirements

3.1 High-Level Goals

The primary goal of this project is to produce a shell scripting language with a strong, static type-and-effect system. This language will be named Marsh, and its type-and-effect system will be based on the Functional Machine Calculus (FMC).

It is unrealistic to expect, given the time and resource constraints of this project, that the final product will be a production-ready shell scripting language. Therefore, it is important to identify high-level goals and non-goals.

Goals This implementation aims to support:

- The basic features of a straight-line programming language, including:
  - String and integer literals
  - Constant and variable declarations
  - Variable assignment
  - Constant and variable usage in expressions
  - Arithmetic operators
- Control flow, by way of:
  - Boolean literals
  - Comparison operators
  - `if/else` expressions
  - `while` loops
- Function definitions and calls.
- Input from the standard input stream.
- Output to the standard output and error streams.
- Generating random numbers.
- Translation into FMC terms for evaluation on a stack machine.
- A type-and-effect system that fully represents both the type and all effects of each component of a program, and rejects incorrectly typed input.

In many cases—particularly in matters of syntax—there are a number of possible approaches to meeting these goals, the differences between which are merely matters of opinion or preference. The process of resolving conflicts between those approaches and choosing a path to follow is therefore often guided by the following constraints:

- Syntactic constructs should be unambiguous to parse.
- Any construct with an equivalent in the FMC should be syntactically as close as possible to its equivalent.

Non-Goals The following are out of scope of this project:

- Types or effects that express a variable number of occurrences of an element, such as "read zero or more strings from the standard input". This requires an extension to the FMC, namely *stream types*, which represent exactly that concept.
- Piping the results of a computation into another computation, along the lines of `@read_strings | @lowercase`, an expression that reads a sequence of strings from the standard input and runs a function `lowercase` on each one.

  Apart from the fact that the type of the `read_strings` function would need to be expressed using a stream type, the type of `lowercase` would require another extension to the FMC, that of the *dual* to stream types.

3.2 Detailed Requirements

The most basic requirements surround the foundational ability to represent and manipulate values.

Requirement 1. The syntax *must* include string, integer and Boolean literals.

Literals are values that appear directly in source code, such as `"Hello, world!"`, `153`, or `false`. This is a basic feature of almost every programming language, and will allow the user to work with values that are statically known before the program is run, without needing to read them from an external location.

Requirement 2. It *must* be possible to perform arithmetic on integer values.

It is also necessary that values be able to be bound to names and stored in variables.

Requirement 3. It *must* be possible to assign names to constant values, and use those names in expressions.

Without the ability to assign a name to a value to refer to later, programs become very difficult to read, placing additional cognitive load on the programmer. In addition, certain other constructs are more difficult to express, such as named function parameters or the use of the same value twice.

Therefore, Marsh should include constant declarations which bind a value to a name, and allow that name to be used in expressions later, where it will be replaced with the value during β-reduction.

Requirement 4. It *should* be possible to assign names to mutable values, update those values, and use their current values in expressions.

One of the advantages of the FMC is that it is capable of expressing not just call-by-name but also call-by-value semantics [2]. Variables will leverage the FMC's *locations* to store a value in a memory cell, which can be accessed and updated after the initial declaration.

Structured control flow is an essential element of a modern programming language. Marsh will aim to include both `if/else` expressions and `while` loops.

Requirement 5. It *must* be possible to conditionally branch based on the value of a Boolean expression.

Requirement 6. The bodies of `if/else` expressions *may* be required to have exactly the same type.

Without support for stream types, for an `if/else` expression must be able to be replaced with one of its two bodies during β-reduction, each of the bodies must express exactly the same type.

Requirement 7. It *should* be possible to repeatedly evaluate a section of code until a Boolean expression evaluates to `false`.

In order to support branching and looping, it is necessary to be able to perform comparisons of values. These comparisons should support not just literals, but the constants described in Requirement 3 and variables in Requirement 4.

Requirement 8. It *should* be possible to compare strings for equality.

Requirement 9. It *should* be possible to compare integers for equality and ordering.

Requirement 10. It *must* be possible to refer to locations unambiguously by name.

Locations are a foundational concept to the FMC, and so use of locations in Marsh should feel natural and seamless. This will be facilitated by dedicated syntax for locations, allowing them to be easily visually distinguished from both identifiers and types.

In particular, it should not be necessary to construct a location by converting it from a string or calling a function.

Requirement 11. Marsh *must* be statically typed.

Requirement 12. Explicit static type annotations *may* be required in some places.

The type of each variable, expression, function, etc. must be statically known. To that end, explicit type annotations may be required where necessary to remove ambiguity. For example, a pop from a location may be required to explicitly specify the type of the popped value, to ensure that any expressions containing that pop have a statically known type.

Chapter 4

Design

4.1 Syntax

Marsh's syntax is inspired primarily by Rust and Haskell. Function definitions begin with a `fn` keyword and must always include a fully specified type. The type of a function includes not only the types of any arguments and its return value, but also any effects the function has.

The example function in Listing 4.1, `print_sum`, takes two arguments, `x` and `y`, both of type `Int`, and has an output effect, `'stdout Int`. This means that the function pushes a single value of type `Int` to the location `'stdout`.

```
fn print_sum :: (x: Int, y: Int) -> ('stdout Int) {
  [x + y]'stdout;
}
```

Listing 4.1: Example Marsh Function (`print_sum`)

4.1.1 Expressions & Statements

Marsh is expression-based, which means that most operations that resolve to a value can be used inside other expressions. For example, as one might expect, a pop from a location—the syntax for which is designed to be similar to the FMC—is an expression that resolves to the popped value. This can be seen in Listing 4.2, where a pop from the `'stdin` location (highlighted in red) is used in a variable declaration, a push expression, and a function call.

```
let input = 'stdin<String>;
['stdin<String>]'stdout;
@say_hello 'stdin<String>;
```

Listing 4.2: Pop Expression Examples

Any expression can be converted into a statement by adding a semicolon after it. This still evaluates the expression, but discards the result. The last line of Listing 4.2 is actually an example of this; anything returned from the call to `say_hello` will be discarded, because the expression has been followed by a semicolon.

This allows for a convenient feature: blocks (curly brace-delimited sections of code, such as function bodies) are also expressions. Their contents must be a sequence of statements, followed by an optional expression. In practice, this means that each line of a block is terminated with a semicolon, but this can be left off of the last line, in which case that line's value becomes the value of the whole block. This can be seen in Listing 4.3, where the function `return_string` returns the `String` it popped from `'stdin` because the lack of a semicolon means its body resolves to the value of its last line, while `discard_string` discards it and returns nothing for the opposite reason.

```
fn return_string :: ('stdin String) -> (String) {
  'stdin<String>
}

fn discard_string :: ('stdin String) -> () {
  'stdin<String>;
}
```

Listing 4.3: Optional Final Expression

A block can be used anywhere that any other kind of expression can be used, as shown in Listing 4.4, where the constant `sum` is initialised to the value of an entire block that pops two `Int`s from `'stdin` and adds them together.

```
let sum = {
  let first = 'stdin<Int>;
  let second = 'stdin<Int>;
  first + second
};
```

Listing 4.4: Constant Initialised With a Block

Some operations in Marsh are always statements. For example, a push to a location does not resolve to a value, and so cannot be used as an expression. Every occurrence of a push must end with a semicolon, and leaving it off results in a syntax error. Constant and variable declarations are also statements, because they do not resolve to a value; if the value of the constant or variable they declare is needed, its name can be used in subsequent expressions.

4.1.2 Locations

Marsh's syntax for locations is inspired by Rust's lifetime syntax; locations consist of an alphanumeric identifier preceded by a single tick (`'`).

Locations are defined fully by their names as they appear in source code, such that two locations with the same name are guaranteed to be equal. This is in contrast to variables, in that two variables with the same name in different scopes may refer to different values. For example, in Listing 4.5, the location `'stdout` that appears in the signature of `foo` is equivalent to the `'stdout` in the signature of `bar`, precisely because they both have the same name.

```
fn foo :: (str: String) -> ('stdout String) {
  [str]'stdout;
}

fn bar :: () -> ('stdout String) {
  ["Hello, world!"]'stdout;
}
```

Listing 4.5: Equivalent Locations

In this implementation, there are four built-in locations accessible to the programmer: `'stdin`, `'stdout`, and `'stderr`, corresponding to the standard Unix input and output streams for a process, and `'rand`, which provides random integer values. These locations are *streams* rather than *stacks*, and this is enforced at runtime; if an attempt is made to push to `'stdin` or `'rand` or pop from `'stdout` or `'stderr`, the program will panic.

However, locations are used in another way in Marsh that is not directly accessible to the programmer: to implement mutable state. Usages of and updates to variable declarations are modelled under the hood as pops from and pushes to unique locations that are not nameable in source code. This will be explained in more detail in Section 4.2.2.

4.1.3 Function Calls

Consider the snippet of Marsh in Listing 4.6. What is the meaning of the usage of `read_int` (highlighted in red)? In other words, what is the value of `something`? Is it a function of type `('stdin Int) -> (Int)`, or a value of type `Int`?

```
fn read_int :: ('stdin Int) -> (Int) {
  'stdin<Int>
}

let something = read_int;
```

Listing 4.6: Ambiguous Usage of `read_int`

The inclusion of effects in Marsh causes this to become ambiguous: the function `read_int` does not require any parameters to be passed to it to run, so nothing distinguishes a call to the function from simply referencing the function itself. Several solutions to this ambiguity were considered.

Postfix Parentheses The solution that appears most straightforward is to require function call syntax to use postfix parentheses containing a comma-separated list of arguments, as in Listing 4.7. This is closest to the syntax used by many imperative languages, such as Rust, and makes it clear that the usage highlighted in red is a function call, while the usage highlighted in blue is a reference to the function itself.

However, this syntax makes partial application less ergonomic, and introduces two possible ways of providing arguments to a function, as illustrated in Listing 4.8:

```
let result = read_int();
let read_int_2 = read_int;
```

Listing 4.7: Required Postfix Parentheses

by listing them inside the parentheses as normal, or by exploiting partial application to first provide one argument to the original function and then providing the other argument to the result. Even if users are unlikely to use the second option, this reduces uniformity.

```
let result_1 = sum(4, 38);
let result_2 = sum(4)(38);
```

Listing 4.8: Exploitation of Partial Application

This also brings the syntax farther away from that of traditional shell scripting languages, where arguments to commands are space-separated and not surrounded by parentheses. Neither of these issues are serious, but are best avoided.

Wrapped References Another option is to require references to functions to be wrapped in a delimiter, such as parentheses or backticks. Unfortunately, this option of requiring "quoting" raises a new issue: how should expressions surrounded by backticks that do not evaluate to a function be handled? It is not possible to check at the parsing stage whether a given arbitrary expression will evaluate to a function, so any incorrect usage of quoting would not be detected until the type-checking pass. While this isn't necessarily a fatal flaw, this was judged likely too confusing for the end user to be a viable option.

Call Operator A third option is to introduce a "call operator", `@`, which must be used whenever a function is called to distinguish that usage from a reference to the function. This transforms the example in Listing 4.7 into the one in Listing 4.9.

```
let result = @read_int;
let read_int_2 = read_int;
```

Listing 4.9: Call Operator

This option is unambiguous, and does not leave open the possibility of providing arguments to functions in more than one way. Therefore, this is the syntax that was implemented.

4.2 Types

Marsh only exposes a subset of the possible types representable using the FMC to the programmer. In particular, access to the main program stack (λ stack) is restricted in order to maintain lexical scoping, to produce correct type judgements, and to prevent the user from interfering with the program's execution.

For example, if Marsh allowed the programmer to interact with the λ stack like any other location, the program in Listing 4.10 would output `"7"`. This is because by pushing an extra argument onto the λ stack, the second argument to the call to `print_sum` is provided manually, despite the line `@print_sum 2` having been judged to have the type `('λ Int) -> ('stdout Int)`. This is only a simple example, but serves to illustrate the point that allowing unrestricted access to the λ stack would violate a number of assumptions about program execution, particularly those made by the type-checker.

```
fn print_sum :: (x: Int, y: Int) -> ('stdout Int) {
    [x + y]'stdout;
}

[5]'λ;
@print_sum 2;
```

Listing 4.10: Interference with the λ Stack

4.2.1 Calling Convention

Arguments in Marsh function calls are provided by value. This means that the expressions making up each of the provided arguments are evaluated, and then their results are pushed to the program stack in reverse order. When the function itself is evaluated, it pops each of those values back off of the program stack.

4.2.2 Variables

Variables are represented using memory cells, which are stacks of depth one [2]. When a variable declaration is encountered, it is translated into a series of terms that evaluate the initial expression, popping it from the program stack, and then push the result to a unique location.

The location used for the memory cell is guaranteed to be unique because it is generated (at type-checking time) from a combination of the variable name and a randomly generated UUID.[1]

Since the type of the variable's initial value and the name of its location are stored in the type-checking environment, it is easy to type-check each update and usage of a variable. Each usage of a variable will later be translated to a pop from that variable's location, so a usage pops the stored type from that location and pushes it to the program stack. Updates are similar, in that they pop a value of the same type as the stored value from the program stack and push it to the variable's location.

Cleanup This representation raises an interesting question. Consider the Marsh code in Listing 4.11. Intuitively, the type of this block should be `() -> ('λ Int)`. But the translation of this block will contain a push to the location used for the

[1]In fact, randomly generated UUIDs are not guaranteed to be unique, but the chance of a collision is so low that they can be assumed to be such.

value of `x` (call it `'x#000`) as a result of the declaration, and a further pop from and push to that location as a result of the access. So the type of this block is actually `() -> ('λ Int, 'x#000 Int)`.

```
{
  var x = 3;
  x
}
```

Listing 4.11: Simple Usage of a Variable

It may not be immediately obvious whether this is a serious problem. In fact, the type `() -> ('λ Int, 'x#000 Int)` *is* an accurate reflection of the effects of this expression. The issue here is a matter of scope: the variable `x` is not accessible outside of the curly braces anyway, so the fact that this expression pushes to it is irrelevant to any code outside.

Not only does this make the type of the expression potentially confusing to the programmer, it causes a concrete problem: the function in Listing 4.12 does not type check! The body of the function should have the type `('stdin Int) -> ('λ Int)`, but is found to be `('stdin Int) -> ('λ Int, 'x#000 Int)`.

```
fn read_add :: ('stdin Int) -> (Int) {
  var input = 'stdin<Int>;
  input = input + 2;
  input
}
```

Listing 4.12: Variable in Function Body

Two potential solutions to this were considered. The first is to translate variable declarations to two terms—a pop from the new variable location to clear it, followed by a push of the initial value—rather than just a push. However, from the perspective of the Functional Abstract Machine (FAM) implementation, this is difficult to handle, because newly created variable locations do not have a value in them, so to allow a pop from that location under these circumstances would result in a messy special case.

The second solution is to insert terms at the end of a block when translating it to clear any locations corresponding to variables that were newly declared inside. The implementation of this feature requires the type-checker to track the names and locations of declared variables in each frame, but does not require any modification of the semantics of the FAM. Therefore, this is the option that was implemented.

4.3 FMC Term Representation

4.3.1 De Bruijn Indices

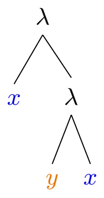
For efficient reduction of terms, it is convenient to use a representation that does not require the renaming of variables in order to avoid unintended variable capture.

De Bruijn described one such representation [9] for the λ-calculus, given by the grammar

$$M, N \quad ::= \quad n \mid \lambda. M \mid MN$$

where $n \in \mathbb{N}$ represents the number of abstractions above a variable n in the syntax tree that must be skipped to find its binder.

For example, the term $\lambda x.\lambda y.x$ is represented using De Bruijn indices by $\lambda.\lambda.1$. This is further illustrated by the syntax tree diagrams in Figure 4.1.


(a) Using Named Variables

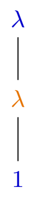
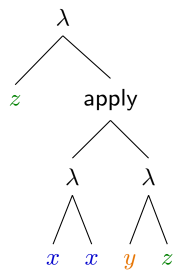

(b) Using De Bruijn Indices

Figure 4.1: Two Representations of $\lambda x.\lambda y.x$

This is unambiguous even in the case of terms containing nested abstractions and applications; for example, the term $\lambda z.(\lambda x.x)(\lambda y.z)$ is represented using De Bruijn indices by $\lambda.(\lambda.0)(\lambda.1)$, again illustrated in Figure 4.2.


(a) Using Named Variables

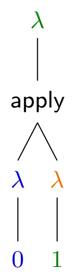

(b) Using De Bruijn Indices

Figure 4.2: Two Representations of $\lambda z.(\lambda x.x)(\lambda y.z)$

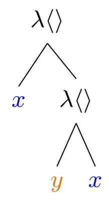
A similar alternative representation for the Functional Machine Calculus is possible.

Definition 1. *The FMC with indexed terms* is given by the grammar

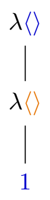
$$M, N \quad ::= \quad \star \mid n. M \mid [N]a. M \mid a\langle\rangle. M$$

where $n \in \mathbb{N}$ represents the number of *pop actions* above the variable n in the syntax tree that must be skipped to find its binder. As in the notation by Barrett, Heijltjes and McCusker for the FMC using named variables, the trailing $.\star$ of a term and the location λ will generally be omitted [2].

The example λ-calculus term from Figure 4.1, $\lambda x.\lambda y.x$, can be represented in the FMC by $\langle x\rangle.\langle y\rangle.x$. This can in turn be represented using indexed terms by $\langle\rangle.\langle\rangle.1$, as illustrated in Figure 4.3.



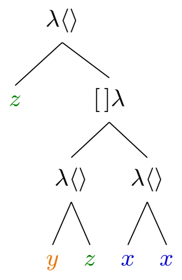
(a) Using Named Variables



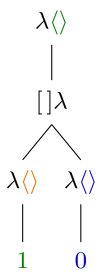
(b) Using Indices

Figure 4.3: Two Representations of $\langle x\rangle.\langle y\rangle.x$

Similarly, the λ-calculus term from Figure 4.2, $\lambda z.(\lambda x.x)(\lambda y.z)$, can be represented in the FMC by $\langle z\rangle.[\langle y\rangle.z].\langle x\rangle.x$. This can again be represented using De Bruijn indices by $\langle\rangle.[\langle\rangle.1].\langle\rangle.0$, as illustrated in Figure 4.4.



(a) Using Named Variables



(b) Using Indices

Figure 4.4: Two Representations of $\langle z\rangle.[\langle y\rangle.z].\langle x\rangle.x$

Substitution and beta-reduction of FMC terms using De Bruijn indices can then be defined, based on the definitions by Barrett, Heijltjes and McCusker for the FMC with named variables [2].

Definition 2. *Incrementing of an indexed FMC term* N^{+i} of a term N by i is defined as the term N where every free variable is incremented by i.

Definition 3. *Capture-avoiding substitution of indexed FMC terms* $\{M/i\}N$ of M for the index i in N is defined as follows:

$$\begin{array}{rcll} \{M/i\}\star & = & \star & \\ \{M/i\}j.\,N & = & j.\,\{M/i\}N & (j<i) \\ \{M/i\}j.\,N & = & N^{+i}.\,\{M/i\}N & (j=i) \\ \{M/i\}j.\,N & = & (j-1).\,\{M/i\}N & (j>i) \\ \{M/i\}[P]a.\,N & = & [\{M/i\}P]a.\,\{M/i\}N & \\ \{M/i\}a\langle\rangle.\,N & = & a\langle\rangle.\,\{M/i+1\}N & \end{array}$$

Definition 4. *Reduction of indexed FMC terms* is by the β-rewrite rule

$$[M]a.\,A_1\ldots A_n.\,a\langle\rangle.\,N \;\rightharpoonup\; A_1\ldots A_n.\,\{M/0\}N$$

where each A_i is an application $[P]b$ or abstraction $b\langle\rangle$ where $b\neq a$.

Chapter 5

Implementation and Testing

5.1 Overview

The structure of the Marsh interpreter can be divided into a series of discrete stages through which source code must be transformed before reaching a point where it can be evaluated to produce a result. These stages are as follows:

- Abstract Syntax Tree (AST) construction
  - Lexing
  - Parsing
- Type checking
- Translation
  - AST types to named FMC terms
  - Named FMC terms to indexed FMC terms
- β-reduction
- Evaluation

This chapter will visit each of these stages in turn, giving a high-level overview of their structure and providing more detail on aspects of the implementation that are of particular complexity or novelty. This also serves as an account of the *process* of implementation, as these components were implemented in the same order.

Implementation Language The entirety of Marsh was implemented in Rust. The combination of Rust's type safety and speed makes it ideal for an interpreter implementation.

In particular, its support for algebraic data types allows for the representation of data with a complex structure (such as the abstract syntax tree) without the possibility of invalid values. The implementation makes heavy use of structural pattern matching on these values, which allows them to be analysed or transformed into new data structures as necessary, with an assurance that all of their possible values are accounted for.

Source Control Throughout the implementation, the source code for this project was kept under `git` source control. The repository was mirrored on more than remote, as well as local copies. In many cases, separate branches were used to house in-progress implementations of various features before being merged back into the `main` branch, in order to provide clean separation between commits and allow for experimentation without affecting other features.

In several cases, having used source control for this project proved to be extremely beneficial. At various points, implementation decisions were revised, and the ability to search back through the change history and roll back certain commits was invaluable.

5.2 Lexing

LALRPOP is an LR(1) parser generator for Rust [20]. This allows a grammar to be specified using a declarative syntax, from which code is generated to parse input. Marsh's syntax is specified using a LALRPOP grammar, a complete copy of which is provided in Section A.1.

Like many parsers, LALRPOP works in two stages: lexing and parsing. The lexer breaks the raw input text up into a series of *tokens*, or *lexemes*, and feeds them to the parser. The parser then analyses this stream of tokens to determine which of many parsing rules are applicable to the input, and thus determine their meaning. This initial tokenisation phase is necessary because despite the fact that the input text may be visually split at word boundaries, there are no boundaries inherent to the input data, and ambiguities in the interpretation of the raw data must be resolved before proceeding.

```
fn print_int :: (
    x: Int
) -> (
    'stdout Int
) {
    [x]'stdout;
}
```

Listing 5.1: Example Function

```
KEYWORD_FN IDENT DOUBLE_COLON
L_PAREN IDENT COLON TYPE
R_PAREN THIN_ARROW L_PAREN
LOCATION TYPE R_PAREN L_CURLY
L_SQUARE IDENT R_SQUARE
LOCATION SEMI R_CURLY
```

Listing 5.2: Equivalent Token Names

For example, Listing 5.1 shows an example Marsh function, `print_int()`. Listing 5.2 contains the names of the equivalent stream of tokens produced when that function is fed through the LALRPOP lexer. Note that this stream of token names is not completely equivalent to the original source code, since the values of each identifier are not included. The actual tokens contain not only the original parsed text but also information about the location in the input where they appeared.

In order to generate this stream of tokens, LALRPOP needs to be told which sequences of characters should be recognised as tokens. In order to so, the grammar file includes a `match` block, an example of which is shown in Listing 5.3. This consists of a series of *match arms*, each of which specifies a regular expression for the lexer to match against input text, producing a token with the corresponding name each time one matches.

The first match arm in Listing 5.3, for example, simply matches the string "fn", which produces a token named `KEYWORD_FN`. The second arm matches strings consisting of a lowercase letter or underscore followed by any additional lowercase letters, numbers, underscores, or tick characters (`'`). Conflicts between match arms are resolved by the order in which they are declared, so the input `fn` would be parsed as `KEYWORD_FN` and not an `IDENT` with the value "fn".

```
match {
    "fn" => KEYWORD_FN,
    r"[a-z_][a-z0-9_']*" => IDENT,
    "{" => L_CURLY,
    r"\s*" => { },
}
```

Listing 5.3: Snippet of Lexer `match` Declaration

5.3 Parsing

After source code has been transformed into a token stream, parsing can take place. In LALRPOP, the grammar is specified by several *productions*, which are rules describing how a particular sequence of tokens should be transformed into an Abstract Syntax Tree (AST) type.

The simplest productions simply recognise a single token, such as the ones for identifiers, type names and locations shown in Listing 5.4. From left to right, each production consists of a name, the AST type it produces, the tokens it recognises, and its action, which consists of the Rust code necessary to produce the specified AST type. For simple rules, this takes the format `name: type = tokens => action;`. Inside the action, the symbol `<>` refers to the entirety of the tokens matched by the pattern.

```
Ident: Ident = IDENT => Ident(names.get_or_intern(<>));
TypeName: TypeName = TYPE => TypeName(names.get_or_intern(<>));
Loc: Loc = LOCATION => Loc(names.get_or_intern(<>));
```

Listing 5.4: Simple Production Rules

However, despite their simplicity, there is a noteworthy detail to the way the generated code for these productions operates. Rather than storing the original identifier, type or location string from source code, the parsed value is *interned*. Interning is an important step in parsing, because it stores the original string in a data structure and allows it to be referred to by a numerical index instead. Interned values can then be compared very cheaply throughout the rest of the progam, rather than needing to perform an expensive string comparison.

The left-hand side pattern of a production can also contain the names of other productions, as in the `PrimaryExpr` production in Listing 5.5. This production, which recognises "primary expressions"—expressions that are either the most basic indivisible unit, such as identifiers, or wrap other expressions in such a way that they can be treated as a single unit—has four arms. The first two refer to other productions, and their actions simply wrap the AST type produced by those productions' actions in a new type. The last two refer to both raw tokens and other productions.

```
PrimaryExpr: Expr = {
    Literal => Expr::Literal(<>),
    Ident => Expr::Ident(<>),
    <loc:Loc> L_ANGLE <ty:Type> R_ANGLE => Expr::Pop { <> },
    L_PAREN <Expr> R_PAREN => <>,
}
```

Listing 5.5: `PrimaryExpr` Production

5.3.1 Possible Alternatives

There are other approaches to parsing than using a parser generator, the most notable among which are *parser combinators* and hand-written parsers.

Parser Combinators Using a parser combinator approach, functions are defined to recognise the structure in small portions of the raw input (analogous to the lexing stage when using a parser generator), and built up into more complex parsers using combinators, which are simply functions that take parsers as input.

For example, the `nom` crate, a Rust parser combinator library, includes combinators such as `alt()`, which tries a second parser if the first fails [27]. This is the equivalent of a production in a grammar with more than one arm. Other combinators include `opt()`, which makes a parser optional, and `many0()`, which applies a parser zero or more times.

However, while parser combinators are arguably more straightforward to use, and may result in more easily understandable code, they suffer from a major drawback: it is easy to run into ambiguities that are difficult to detect and difficult to resolve without major restructuring of the code. Parser generators are often able to automatically detect ambiguities in a grammar, while in the case of parser combinators ambiguities frequently result in infinite loops with no clear sign of their cause.

The choice to use a parser generator partly on this basis proved beneficial. While ambiguities did arise between productions at various points in the implementation of Marsh, in all cases they were automatically detected and trivial to resolve, usually simply requiring an arm of a production to be separated into a new rule. In fact, resolving these problems often revealed that the root cause of the ambiguity was that the original grammar was attempting to be too permissive, and the corrected version was more in line with the intended syntax.

Hand-Written Parsers It is also relatively common for programming language implementations to take the approach of writing a custom lexer and parser from scratch. However, this is very time consuming and thus is normally seen only in mature implementations. It was judged that a parser generator was a more effective use of time for this implementation, especially since LALRPOP allows a custom lexer to be used in place of its built-in lexer if necessary.

5.3.2 Abstract Syntax Tree

The AST is a complete representation of all the essential information about the structure of parsed code, with extraneous information (such as whitespace and comments) removed. It consists of both `struct`s, which are product types, and `enum`s, which are sum types.

For example, as the literal values that can appear in Marsh source code are each one of a defined set of options (the unit value, integers, booleans, strings, or paths), the `Literal` AST type is an `enum` (Listing 5.6). Each instance of a `Literal` can be exactly one of its five variants, four of which contain the original value of the literal as it was parsed.

`Expr` is also an `enum`, which represents the possible kinds of expression: a literal, an identifier, a pop from a location, a call to a function, an application of several possible kinds of operator, an if-else expression, or a curly-brace-delimited block. Likewise for `Stmt`, which represents statements: a `let` (constant) declaration, a variable declaration, a variable assignment, a push to a location, or an expression whose result is discarded.

```
pub enum Literal {
    Unit,
    Int(isize),
    Bool(bool),
    String(String),
    Path(PathBuf),
}
```

Listing 5.6: `Literal` AST Type

```
pub struct Func {
    pub name: Ident,
    pub locs: Vec<Loc>,
    pub eff_in: Vec<Effect>,
    pub args: Vec<Arg>,
    pub eff_out: Vec<Effect>,
    pub ret: Option<Type>,
    pub body: Block,
}
```

Listing 5.7: `Func` AST Type

In contrast, since function declarations always have the same overall structure, the AST type `Func` is a `struct`, as shown in Listing 5.7. As a result, all of the fields in each instance of a `Func` must have a value—although some of those values may not contain much of interest, as in `eff_in` or `args`, which could be empty lists, or `ret`, which could be `Option::None`.[1]

It's worth also noting the way that types and effects are represented. `Type` is an `enum` (reproduced in Figure 5.8), because a type could either be simply a type name—such as `Int`—or an arrow type, representing a term that can be evaluated and possibly performs effects in the process—such as `(String) -> ('stdout String)`. Meanwhile, `Effect` is a `struct` containing a type together with a location. By nesting these two AST types within each other, it is possible to represent types of unlimited complexity.

[1]The `Option` enum is Rust's equivalent of Haskell's `Maybe` monad. Since Rust deliberately does not include nullable pointers in safe code, values that are allowed to either contain either nothing or some type `T` must use the type `Option<T>`. This enum has two cases, `Some` (equivalent to `Just`) and `None` (equivalent to `Nothing`), along with a rich API of combinators and transformers.

```
pub enum Type {
    Unit,
    Base {
        name: TypeName,
    },
    Arrow {
        eff_in: Vec<Effect>,
        ty_in: Vec<Type>,
        eff_out: Vec<Effect>,
        ty_out: Box<Type>,
    },
}
```

Listing 5.8: `Type` AST Type

```
pub struct Effect {
    pub loc: Loc,
    pub ty: Type,
}
```

Listing 5.9: `Effect` AST Type

A complete copy of each of the AST types' definitions is provided in Section A.2.

5.4 Type Checking

After source code has been parsed and transformed into an AST, the next necessary step is to perform type-checking. Many of the details of this process are informed by the knowledge of the eventual FMC term representation that each AST type will be translated to, so this section will not confine itself to exclusively discussing type-checking without mention of the following translation step.

Type-checking is performed for each AST type through an implementation of the `TypeCheck` trait, reproduced in Listing 5.10. Each implementation consumes an AST type and produces a typed version, such as `TypedExpr` for `Expr`. These new versions are referred to as "typed" both because their contents have been type-checked and a valid type derivation produced, and because they implement the trait `Typed`, which requires them to implement a method returning the type of one of their usages. For example, a string literal would return the type `() -> ('λ String)`, because a usage of a string literal pushes a value of type `String` to the λ stack.

As a part of the process of type-checking, an environment (`Env`) is maintained, which contains a tree of `Frame`s. As the types of the values stored in various identifiers are determined, they are stored in frames in the environment according to the scope in which they appear.

```
pub trait TypeCheck {
    type Output: Typed;

    fn check(
        self, env: &mut Env<'_>, frame: Uuid
    ) -> Result<Self::Output, TypeError>;
}
```

Listing 5.10: `TypeCheck` Trait

5.5 Translation

5.5.1 AST to FMC Terms

The next step is to translate the type-checked AST into FMC terms. Similarly to type-checking, this is performed through an implementation of the `Translate` trait for each typed AST type (reproduced in Listing 5.11).

```
pub trait Translate {
    type Translated;

    fn translate(self, env: &mut Env<'_>) -> Self::Translated;
}
```

Listing 5.11: `Translate` Trait

If/Else `if/else` expressions presented a problem in terms of translation. Barrett, Heijltjes and McCusker describe an example machine transition for an `if` builtin as follows [2]:

$$\frac{(\ S_A\ ;\ S_\lambda \cdot P \cdot N \cdot \top\ ,\ \mathsf{if}.\,M\ )}{(\ S_A\ ;\ S_\lambda \cdot N\ ,\ M\ )} \qquad \frac{(\ S_A\ ;\ S_\lambda \cdot P \cdot N \cdot \bot\ ,\ \mathsf{if}.\,M\ )}{(\ S_A\ ;\ S_\lambda \cdot P\ ,\ M\ )}$$

Unfortunately, the bodies of each arm of `if/else` expressions in Marsh may consist of a variable number of terms. Due to the combination of this and the flat representation Marsh uses for sequences of terms, additional markers are needed to make this transition unambiguous with respect to how many terms should be popped from S_λ. Therefore, Marsh instead uses the following machine transitions:

$$\frac{(\ S_A\ ;\ S_\lambda \cdot \mathsf{endif} \cdot P \cdot \mathsf{else} \cdot N \cdot \mathsf{then} \cdot \top \cdot \mathsf{if}\ ,\ M\ )}{(\ S_A\ ;\ S_\lambda \cdot N\ ,\ M\ )}$$

$$\frac{(\ S_A\ ;\ S_\lambda \cdot \mathsf{endif} \cdot P \cdot \mathsf{else} \cdot N \cdot \mathsf{then} \cdot \bot \cdot \mathsf{if}\ ,\ M\ )}{(\ S_A\ ;\ S_\lambda \cdot P\ ,\ M\ )}$$

5.5.2 Named to Indexed Terms

This requires an intermediate operation: identification of free variables in named terms.

Free Variable Identification For identification of free variables, the `Free` trait is implemented for `Vector<NamedTerm>`. This implementation is streamlined by the use of a `BTreeSet`.

`BTreeSet` is a Rust standard library data structure that is similar to a hash set, but uses a B-tree for its underlying storage. In cases where the stored type implements `Ord`, which allows it to be compared for ordering with other instances of the same type, this storage is often faster and more compact than that of a hash set. Helpfully, since identifiers are interned rather than containing raw string values and are therefore represented using numerical identifiers, this ordering comparison is very fast.

Each time a term of the `Value` variant containing a variable is encountered at the head of a list of terms, that variable is added to the set of free variables returned by a recursive call to `Free::free()` on the tail. Likewise, each time the head of the list is a `Pop` term, the variable bound by that term is removed from the set returned by the recursive call. The result is a set of variable names encountered in terms for which no binder was found.

5.6 Beta Reduction

5.6.1 Named Terms

Before implementing β-reduction using indexed terms as described in Section 4.3.1, an initial implementation using named terms was deemed a possible intermediate step. This relies on the ability to rename free variables in terms in order to perform capture-avoiding substitution.

Unfortunately, rather than an appropriate intermediate step, β-reduction of terms containing named variables is a separate, and possibly more difficult, feature. In particular this is due to the requirement to implement a method of checking terms for α-equivalence in order to write automated tests for this functionality. Rather than implementing α-equivalence, which is unnecessary for terms using variable indices, directly implementing reduction for indexed terms was determined to be a more effective use of time.

5.6.2 Indexed Terms

First, named terms created by the translation from AST types described in Section 5.5.1 must be translated to indexed terms, as described in Section 5.5.2. After this, β-reduction relies on the implementation of two more intermediate operations: free variable lifting and substitution of indexed terms. The implementation of these operations is straightforward following Definitions 2 and 3, and is expressed through implementations of two traits `Lift` and `Substitute` for the type `Vector<IndexedTerm>`.

5.7 Evaluation

Evaluation of FMC terms is the final step. This relies on an implementation of the Functional Abstract Machine [2], which consists of two key components:

- A `HashMap` of “memories”, each of which is an implementation of the `Memory` trait, reproduced in Listing 5.12. The keys in this hash map are the names of the locations each memory represents.

- A `transition()` method, which takes a list of indexed terms as input and attempts to perform a machine transition [2], updating the state of the machine and returning the new list of terms that results.

```
pub trait Memory: Debug {
    /// Attempt to push to this memory.
    fn push(&mut self, term: IndexedTerm) -> Result<(), MemoryError>;

    /// Attempt to pop from this memory.
    fn pop(&mut self) -> Result<IndexedTerm, MemoryError>;
}
```

Listing 5.12: `Memory` Trait

There are six implementations of `Memory`, many of which have unique semantics: `Lambda`, `Stdin`, `Stdout`, `Stderr`, `Rand`, and `Cell`.

Lambda Memory The `Lambda` memory represents the main program stack as described by Barrett, Heijltjes and McCusker [2]. Terms of any type can be both pushed to and popped from this memory.

Stdin The `Stdin` memory represents the standard input stream to the process. This is a stream, allowing only pops and not pushes. Each individual value popped from this stream is a line provided to the input stream, and pops from this location block until such a line is provided.

Stdout & Stderr These locations correspond to the standard output and standard error streams of the process. Like `Stdin`, these are streams, but only allow pushes and not pops. Each value pushed to this location is printed with a newline after it.

Rand This location is a stream, allowing only pops. Each popped value is a signed integer provided by a random number generator.

Cell `Cell` is a stack capable of storing either zero or one value at any given time. Pops will fail if the cell is empty, and pushes will fail if it is full. Multiple instances of this memory can exist at the same time under different names inside the FAM, created in response to the declaration of mutable variables.

5.8 Testing

Each of the major stages of processing is covered by a suite of automated tests. These tests automatically set up input data, such as raw input strings for the parsing stage, untyped AST representations for the type-checking stage, and typed AST representations for the translation stage. This input data is then run through the processing code and checked against an expected value at the end.

Automated tests are in most cases implemented individually for each variant of the types in question—such as one test for each type of expression that it type-checks correctly when given valid input. Listing 5.13 is an example of one of these tests.

```rust
#[test]
fn type_pop_expr() {
    rodeo! {
        names = {
            string: TypeName = "String",
            stdin: AstLoc = "'stdin",
        }
    }

    let pop_expr = crate::parse::parse_with::<Expr>("'stdin<String>",
↪   &mut names)
        .expect("able to parse a pop expression");

    let mut env = Env::new(&mut names);

    assert_eq!(
        pop_expr
            .check(&mut env, Frame::global_id())
            .expect("able to type-check an expression that pops a
↪   String")
            .ty(),
        &Arrow::default()
            .with_left(stdin, string.clone())
            .with_right(Loc::Lambda, string),
        "expression that pops a String type-checks to the correct type"
    );
}
```

Listing 5.13: Test of Type-Checking an `Expr::Pop`

Chapter 6

Conclusions

6.1 Results

The implemented interpreter can type-check and evaluate simple programs, including those consising of multiple function definitions and branching with `if/else` expressions. The language these programs are written in is strongly and statically typed, accurately tracks the effects of each expression, and compiles down to a sequence of terms that can be significantly simplified through β-reduction before being executed on a stack machine.

Listing 6.1 is an example of a higher-order function written in Marsh, which takes a function and an argument and applies the function to the argument. Listing 6.2 is an example of a function that prints a random number, with both the non-determinism and output effect statically type-checked and explicitly annotated.

```
fn call_on :: (f: (Int) -> (Int), x: Int) -> (Int) {
    @f x
}
```

Listing 6.1: Example Higher-Order Function

```
fn print_rand :: ('rand Int) -> ('stdout Int) {
    ['rand<Int>]'stdout;
}
```

Listing 6.2: Non-Determinism and Output

This implementation represents a proof of concept that a high level language can be built to be compatible with the Functional Machine Calculus (FMC) while still retaining its imperative nature, and that the FMC provides a basis for a type system that can express the effects performed by even high-level structured programs, unlike the type systems of other shell scripting languages.

We also introduce a representation of FMC terms that uses indexed rather than named variables for efficient reduction of terms without the requirement for renaming, based on the representation originally proposed by de Bruijn [9].

6.2 Future Work

Several notable features are required for Marsh to become a truly useful scripting language.

Loops Due to time constraints, a looping construct was not implemented. Despite the limitation of a lack of a representation for a variable number of occurrences of a type, it is theoretically possible to perform type-checking of a `while` loop by requiring that the type of the body composes with itself.

External Commands Part of the power of a shell scripting language is the ability to make use of external commands to perform various operations. Despite the fact that external commands not written in Marsh would not be able to be type-checked, it may be possible to treat calls to them as falling under a blanket type such as `('stdin String*) -> ('stdout String*)`, where `String*` is a stream type representing "zero or more `String`s".

Pipes The ability to pipe the results of one operation into another is a very ergonomic feature of shell languages like Bash. While this has can be modelled as a monadic operation in lazy functional languages, to represent this in Marsh would require additional extensions to the FMC.

6.3 Critique

Despite attempts to limit its scope, this project remains an ambitious endeavour, and suffers as a result. An approach more focused on, for example, an end-to-end implementation of the features required to support one particular example program may have been more successful. Another alternative may be a more theoretical examination of the extensions to the FMC required to support the full feature set of traditional shell scripting languages, such as stream types for piping.

On the other hand, despite its shortcomings, this project represents a novel demonstration of the potential practical applications of the FMC. The codebase is of high quality, as it is fully documented, includes automated tests, and is designed in such a way as to be extensible. As a result of being implemented in Rust, adding new AST types is straightforward and does not incur a risk of missed edge cases as a result. This implementation represents a promising foundation for future extensions to support more features.

Chapter 7

Bibliography

[1] Apple Inc. *Use zsh as the default shell on your Mac.* 28th Jan. 2020. URL: https://support.apple.com/en-us/HT208050 (visited on 16/11/2021).

[2] Chris Barrett, Willem Heijltjes and Guy McCusker. “The Functional Machine Calculus”. Submitted. 2022.

[3] Robert L. Bocchino et al. “A Type and Effect System for Deterministic Parallel Java”. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications.* OOPSLA ’09. Orlando, Florida, USA: Association for Computing Machinery, 2009, pp. 97–116. ISBN: 9781605587660. DOI: 10.1145/1640089.1640097.

[4] Digital Equipment Corporation. *DEC System 10 Operating System Commands Manual.* Digital Equipment Corporation. Maynard, M.A., 1977. URL: http://bitsavers.org/pdf/dec/pdp10/TOPS10/AA-0916C-TB_DEC10_Operating_Systems_Command_Manual_Ver_6_03_Aug77.pdf.

[5] Digital Equipment Corporation. *TOPS-10 Monitor Internals.* Revision 6. Digital Equipment Corporation. Bedford, M.A., 1980. URL: http://bitsavers.org/pdf/dec/pdp10/TOPS10_monitorInternalsCourse/EY-CD150-HO-006_monInternal.pdf.

[6] *CVE-2014-6271.* Available from MITRE, CVE-ID CVE-2014-6271. Dec. 2014. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271 (visited on 16/11/2021).

[7] Howard Dahdah. *The A-Z of Programming Languages: Bourne shell, or sh. An in-depth interview with Steve Bourne, creator of the Bourne shell, or sh.* URL: https://web.archive.org/web/20100111203059/http://www.computerworld.com.au/article/279011/-z_programming_languages_bourne_shell_sh/ (visited on 04/04/2022).

[8] A. Danesh and M. Jang. *Mastering Linux.* Wiley, 2006. ISBN: 9780782152777. URL: https://books.google.co.uk/books?id=tIjrVYbZmUAC.

[9] Nicolaas Govert de Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0.

[10] *Fish Shell. Finally, a command line shell for the 90s.* URL: `https://fishshell.com/` (visited on 16/11/2021).

[11] Cormac Flanagan and Shaz Qadeer. "A Type and Effect System for Atomicity". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* PLDI '03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 338–349. ISBN: 1581136625. DOI: `10.1145/781131.781169`.

[12] Isaac Oscar Gariano, James Noble and Marco Servetto. "Call$\mathcal{E}$: An Effect System for Method Calls". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Onward! 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 32–45. ISBN: 9781450369954. DOI: `10.1145/3359591.3359731`.

[13] Carlo Ghezzi. *Programming language concepts.* eng. New York: Wiley, 1982. ISBN: 047186482X.

[14] Paola Giannini, Marco Servetto and Elena Zucca. "A Type and Effect System for Uniqueness and Immutability". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing.* SAC '18. Pau, France: Association for Computing Machinery, 2018, pp. 1038–1045. ISBN: 9781450351911. DOI: `10.1145/3167132.3167245`.

[15] David K. Gifford and John M. Lucassen. "Integrating Functional and Imperative Programming". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming.* LFP '86. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1986, pp. 28–38. ISBN: 0897912004. DOI: `10.1145/319838.319848`.

[16] P. Z. Ingerman. "Thunks: A Way of Compiling Procedure Statements with Some Comments on Procedure Declarations". In: *Commun. ACM* 4.1 (Jan. 1961), pp. 55–58. ISSN: 0001-0782. DOI: `10.1145/366062.366084`.

[17] *Ion.* URL: `https://gitlab.redox-os.org/redox-os/ion` (visited on 11/11/2021).

[18] *Ion Documentation.* URL: `https://doc.redox-os.org/ion-manual/introduction.html` (visited on 24/11/2021).

[19] *jq. Command-line JSON processor.* URL: `https://github.com/stedolan/jq` (visited on 16/11/2021).

[20] *LALRPOP. LR(1) parser generator for Rust.* URL: `https://github.com/lalrpop/lalrpop` (visited on 14/04/2022).

[21] P. J. Landin. "Correspondence between ALGOL 60 and Church's Lambda-Notation: Part I". In: *Commun. ACM* 8.2 (Feb. 1965), pp. 89–101. ISSN: 0001-0782. DOI: `10.1145/363744.363749`.

[22] J. M. Lucassen and D. K. Gifford. "Polymorphic Effect Systems". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. ISBN: 0897912527. DOI: `10.1145/73560.73564`.

[23] Magnus Madsen and Jaco van de Pol. "Polymorphic Types and Effects with Boolean Unification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: `10.1145/3428222`.

[24] Sven Mascheck. *The Traditional Bourne Shell Family. History and Development.* 16th Sept. 2019. URL: `https://www.in-ulm.de/~mascheck/bourne/index.html` (visited on 11/11/2021).

[25] J. R. Mashey. "Using a Command Language as a High-Level Programming Language". In: *Proceedings of the 2nd International Conference on Software Engineering.* ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 169–176.

[26] Eugenio Moggi. "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science.* Pacific Grove, California, USA: IEEE Press, 1989, pp. 14–23. ISBN: 0818619546.

[27] *nom. Rust parser combinator framework.* URL: `https://github.com/Geal/nom` (visited on 05/05/2022).

[28] *Nu Book.* URL: `https://www.nushell.sh/book` (visited on 11/11/2021).

[29] *Nushell. A new type of shell.* URL: `https://github.com/nushell/nushell` (visited on 11/11/2021).

[30] Simon L. Peyton Jones and Philip Wadler. "Imperative Functional Programming". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 71–84. ISBN: 0897915607. DOI: `10.1145/158511.158524`.

[31] Louis Pouzin. "RUNCOM: A Macro-Procedure Processor for the 636 System". In: *Multics Design Notebook, Section V* (1965). URL: `https://people.csail.mit.edu/saltzer/Multics/Multics-Documents/MDN/MDN-5.pdf`.

[32] Louis Pouzin. *The Origin of the Shell.* 25th Nov. 2000. URL: `https://www.multicians.org/shell.html` (visited on 16/11/2021).

[33] Louis Pouzin. "The SHELL: A global tool for calling and chaining procedures in the system". In: *Multics Design Notebook, Section IV* (1965). URL: `https://people.csail.mit.edu/saltzer/Multics/Multics-Documents/MDN/MDN-4.pdf`.

[34] Ellie Quigley. *Linux shells by example.* eng. 1st edition. Upper Saddle River, N.J.: Prentice Hall PTR, 2000.

[35] Lukas Rytz, Nada Amin and Martin Odersky. "A Flow-Insensitive, Modular Effect System for Purity". In: *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs.* FTfJP '13. Montpellier, France: Association for Computing Machinery, 2013. ISBN: 9781450320429. DOI: `10.1145/2489804.2489808`.

[36] *ShellCheck.* URL: `https://github.com/koalaman/shellcheck` (visited on 11/11/2021).

[37] David A. Spuler and A. S. M. Sajeev. "Abstract Compiler Detection of Function Call Side Effects". In: (1994).

[38] Philip Wadler. “Comprehending Monads”. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP ’90. Nice, France: Association for Computing Machinery, 1990, pp. 61–78. ISBN: 089791368X. DOI: `10.1145/91556.91592`.

[39] Philip Wadler. “The Essence of Functional Programming”. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’92. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1992, pp. 1–14. ISBN: 0897914538. DOI: `10.1145/143165.143169`.

[40] *Why you shouldn’t parse the output of ls(1)*. URL: `https://mywiki.wooledge.org/ParsingLs` (visited on 24/11/2021).

Appendix A

Selected Source Code

A.1 LALRPOP Grammar

```
use crate::ast::*;
use lasso::Rodeo;
use tap::Pipe;

use std::path::PathBuf;

grammar<'env>(
    names: &'env mut Rodeo,
);

match {
    "fn" => KEYWORD_FN,
    "let" => KEYWORD_LET,
    "var" => KEYWORD_VAR,
    "if" => KEYWORD_IF,
    "else" => KEYWORD_ELSE,

    "true" => TRUE,
    "false" => FALSE,

    r"[a-z_][a-z0-9_']*" => IDENT,
    r"[A-Z][a-zA-Z0-9]*" => TYPE,
    r"'[a-z][a-z0-9_]*" => LOCATION,

    r"-?0b[01]+" => NUM_BIN,
    r"-?0o[0-7]+" => NUM_OCT,
    r"-?[0-9]+" => NUM_DEC,
    r"-?0x[0-9a-fA-F]+" => NUM_HEX,

    r#""[^"]*""# => STRING,
    r#"\.\.?(:?/(:?\.\.?|[a-zA-Z0-9_-]+))+(:?\.[a-zA-Z0-9_-]+)?"# =>
↪   PATH,

    "::(" => TURBO,

    "->" => THIN_ARROW,
    "=>" => FAT_ARROW,

    "==" => DOUBLE_EQUALS,
    "!=" => EXCL_EQUALS,
    "<=" => LESS_EQUALS,
    ">=" => GREATER_EQUALS,

    "," => COMMA,
    "." => DOT,
    ";" => SEMI,
    "/" => SLASH,
    "@" => AT,
    "*" => STAR,
    "+" => PLUS,
    "-" => DASH,

    "(" => L_PAREN,
    ")" => R_PAREN,
    "{" => L_CURLY,
    "}" => R_CURLY,
    "[" => L_SQUARE,
    "]" => R_SQUARE,

    r"\s*" => { }, // Skip whitespace
    r"#[^\n\r]*[\n\r]*" => { }, // Skip # comments
} else {
    "::" => DOUBLE_COLON,
} else {
    ":" => COLON,

    "=" => EQUALS,
    "!" => EXCL,
    "<" => L_ANGLE,
    ">" => R_ANGLE,
}

// Parse a list of `T`s separated by `S`s, with an optional trailing
↪   `S`.
#[inline]
Star<T, S>: Vec<T> = {
    <mut v:(<T> S)*> <e:T?> => match e {
        None => v,
        Some(e) => {
            v.push(e);
            v
        }
    }
}

// Parse a list of at least one `T` separated by `S`s, with an optional
↪   trailing
// `S`.
#[inline]
Plus<T, S>: Vec<T> = {
    <mut v:(<T> S)*> <e:T> S? => {
        v.push(e);
        v
    }
}

// A program consists of a sequence of zero or more items.
pub Program: Program = Item*;

// Items are anything that can appear at the top level of a program:
↪   function
// definitions or statements.
pub Item: Item = {
    <Func> => Item::Func(box <>),
    <Stmt> => Item::Stmt(box <>),
}

// A function definition, e.g.:
//
// fn read_add :: ('stdin Int, x: Int, y: Int) -> ('stdout Int, Int) {
//     let result = 'stdin<Int> + x + y;
//     [result]'stdout;
//     result
// }
pub Func: Func = {
    KEYWORD_FN <name:Ident> DOUBLE_COLON
        <locs:(<Plus<Loc, COMMA>> FAT_ARROW)?>
        <func_in:FuncIn>
        THIN_ARROW <func_out:FuncOut>
        <body:Block>
    => Func {
        name,
        locs: locs.unwrap_or_default(),
        eff_in: func_in.0,
        args: func_in.1,
        eff_out: func_out.0,
        ret: func_out.1,
        body,
    },
}

// The inputs to a function, consisting of a comma-separated list of
↪  effects
// followed by arguments, all surrounded by parentheses.
FuncIn: (Vec<Effect>, Vec<Arg>) = {
    L_PAREN <Star<Effect, COMMA>> <Star<Arg, COMMA>> R_PAREN => (<>),
}

// An argument to a function is a name and a type.
pub Arg: Arg = {
    <name:Ident> COLON <ty:Type> => Arg { <> },
}

// The outputs of a function, consisting of a comma-separated list of
↪  effects
// followed by a single optional return type.
FuncOut: (Vec<Effect>, Option<Type>) = {
    L_PAREN <effects:Star<Effect, COMMA>> <ty:Type?> R_PAREN => (
        effects,
        ty.and_then(|t| if t == Type::Unit { None } else { Some(t) })
    ),
}

// An effect is a location and a type.
pub Effect: Effect = {
    <loc:Loc> <ty:Type> => Effect { <> },
}

// A block is a curly-brace-delimited sequence of statements, followed
↪  by an
// optional terminal expression, which becomes the value of the block if
↪
// present.
pub Block: Block = {
    L_CURLY <stmts:Stmt*> <expr:Expr?> R_CURLY => Block { <> },
}

// A statement is anything that does not resolve to a value.
pub Stmt: Stmt = {
    <Expr> SEMI => Stmt::Expr(<>),
    KEYWORD_LET <name:Ident> EQUALS <val:Expr> SEMI => Stmt::Let { <> },
    KEYWORD_VAR <name:Ident> EQUALS <val:Expr> SEMI => Stmt::Var { <> },
    <name:Ident> EQUALS <val:Expr> SEMI => Stmt::Assign { <> },
    L_SQUARE <expr:Expr> R_SQUARE <loc:Loc> SEMI => Stmt::Push { <> },
}

// An expression is something that does resolve to a value. This is
↪   split up
// among many levels, some of which are not currently fleshed out, in
↪   order to
// encode precedence and avoid parsing ambiguities.
pub Expr: Expr = {
    EqExpr,
    Block => Expr::Block(box <>),
    KEYWORD_IF <cond:EqExpr> <then:Block> KEYWORD_ELSE <else_:Block> =>
↪   Expr::IfElse {
        cond: box cond,
        then: box then,
        else_: box else_,
    },
}

// Expressions at the "equality" precedence level.
EqExpr: Expr = {
    RelExpr,
    <lhs:EqExpr> <op:EqOp> <rhs:RelExpr> => Expr::Equality(box lhs, op,
↪   box rhs),
}

EqOp: EqualityOp = {
    DOUBLE_EQUALS => EqualityOp::Eq,
    EXCL_EQUALS => EqualityOp::Neq,
}

// Expressions at the "relational" precedence level.
RelExpr: Expr = {
    AddExpr,
    <lhs:RelExpr> <op:RelOp> <rhs:AddExpr> => Expr::Relational(box lhs,
↪   op, box rhs),
}

RelOp: RelationalOp = {
    L_ANGLE => RelationalOp::Lt,
    R_ANGLE => RelationalOp::Gt,
    LESS_EQUALS => RelationalOp::Leq,
    GREATER_EQUALS => RelationalOp::Geq,
}
```

```
// Expressions at the "additive" precedence level.
AddExpr: Expr = {
    MultExpr,
    <lhs:AddExpr> <op:AddOp> <rhs:MultExpr> => Expr::Arithmetic(box lhs,
↪   op, box rhs),
}

// Binary arithmetic operations at the "additive" precedence level.
AddOp: ArithmeticOp = {
    PLUS => ArithmeticOp::Add,
    DASH => ArithmeticOp::Sub,
}

// Expressions at the "multiplicative" precedence level.
MultExpr: Expr = {
    UnaryExpr,
    <lhs:MultExpr> <op:MultOp> <rhs:UnaryExpr> => Expr::Arithmetic(box
↪   lhs, op, box rhs),
}

// Binary arithmetic operations at the "multiplicative" precedence
↪   level.
MultOp: ArithmeticOp = {
    STAR => ArithmeticOp::Mul,
    SLASH => ArithmeticOp::Div,
}

// Expressions at the "unary" precedence level.
UnaryExpr: Expr = {
    CallExpr,
    <operator:UnaryOp> <operand:CallExpr> => Expr::Unary(operator, box
↪   operand),
}

UnaryOp: UnaryOp = {
    EXCL => UnaryOp::Not,
}

// Parse an expression at the "call" precedence level.
CallExpr: Expr = {
    SimpleCallExpr,
    AT <callee:PrimaryExpr> TURBO <locs:Plus<Loc, COMMA>> R_PAREN
↪   <args:SimpleCallExpr+> => Expr::Call {
        callee: box callee,
        locs,
        args,
    },
    AT <callee:PrimaryExpr> <args:SimpleCallExpr+> => Expr::Call {
        callee: box callee,
        locs: vec![],
        args,
    },
}

// A SimpleCallExpr is an unparenthesized call that does not include any
↪
// arguments after it. It's necessary to distinguish between this and a
↪   CallExpr
// for reasons of unambiguity, given this is an LR(1) grammar. These can
↪   be
// thought of as just two more cases of `CallExpr`.
SimpleCallExpr: Expr = {
    PrimaryExpr,
    AT <callee:PrimaryExpr> TURBO <locs:Plus<Loc, COMMA>> R_PAREN =>
↪   Expr::Call {
        callee: box callee,
        locs,
        args: vec![],
    },
    AT <callee:PrimaryExpr> => Expr::Call {
        callee: box callee,
        locs: vec![],
        args: vec![],
    },
}

// Parse a "primary expression", which is the basic grouping of
↪   expressions: it
// is either an indivisible unit like a literal or identifier, or it is
↪   a
// parenthesised expression, which can be *treated* as indivisible.
PrimaryExpr: Expr = {
    Literal => Expr::Literal(<>),
    Ident => Expr::Ident(<>),
    <loc:Loc> L_ANGLE <ty:Type> R_ANGLE => Expr::Pop { <> },
    L_PAREN <Expr> R_PAREN => <>,
}

// Parse a type, interning the names of any base types in the names
↪   Rodeo.
pub Type: Type = {
    L_PAREN R_PAREN => Type::Unit,
    <name:TypeName> => Type::Base { <> },
    L_PAREN
```

```
        <eff_in:Star<Effect, COMMA>>
        <ty_in:Star<Type, COMMA>>
    R_PAREN
    THIN_ARROW
    L_PAREN
        <eff_out:Star<Effect, COMMA>>
        <ty_out:Type?>
    R_PAREN => Type::Arrow {
        eff_in,
        ty_in,
        eff_out,
        ty_out: box ty_out.unwrap_or(Type::Unit),
    },
}

// Parse an identifier, interning the value in the names Rodeo.
pub Ident: Ident = IDENT => Ident(names.get_or_intern(<>));

// Parse a type name, interning the value in the names Rodeo.
pub TypeName: TypeName = TYPE => TypeName(names.get_or_intern(<>));

// Parse a location name, interning the value in the names Rodeo.
pub Loc: Loc = LOCATION => Loc(names.get_or_intern(<>));

// Parse a literal value.
pub Literal: Literal = {
    L_PAREN R_PAREN => Literal::Unit,
    NUM_BIN => isize::from_str_radix(&<>[2..], 2)
        .expect("parsed binary literal can be converted to an isize")
        .pipe(Literal::Int),
    NUM_OCT => isize::from_str_radix(&<>[2..], 8)
        .expect("parsed octal literal can be converted to an isize")
        .pipe(Literal::Int),
    NUM_DEC => isize::from_str_radix(<>, 10)
        .expect("parsed decimal literal can be converted to an isize")
        .pipe(Literal::Int),
    NUM_HEX => isize::from_str_radix(&<>[2..], 16)
        .expect("parsed decimal literal can be converted to an isize")
        .pipe(Literal::Int),
    TRUE => Literal::Bool(true),
    FALSE => Literal::Bool(false),
    STRING => <>[1..(<>.len() - 1)].to_owned().pipe(Literal::String),
    PATH => PathBuf::from(<>).pipe(Literal::Path),
}
```

A.2 Abstract Syntax Tree

```
//! Abstract syntax tree representation of Marsh code.

use std::{fmt, path::PathBuf};

use lasso::Spur;

pub mod reconstruct;
pub mod resolve;

/// A program consists of a sequence of [`Item`]s.
pub type Program = Vec<Item>;

/// An item is anything that can appear at the top level of a file.
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Item {
    /// A function definition at the top level of a file.
    Func(Box<Func>),
    /// A statement item is simply a [`Stmt`] at the top level of a
↪   file.
    Stmt(Box<Stmt>),
}

/// A function definition, including its [`Arg`]s, return [`Type`], and
↪   body [`Block`].
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct Func {
    /// The name of the function.
    pub name: Ident,
    /// The location parameters of the function.
    pub locs: Vec<Loc>,
    /// The input effects of the function.
    pub eff_in: Vec<Effect>,
    /// The arguments to the function.
    pub args: Vec<Arg>,
    /// The output effects of the function.
    pub eff_out: Vec<Effect>,
    /// The return type of the function.
    pub ret: Option<Type>,
    /// The block that makes up the body of the function.
    pub body: Block,
}

/// A function argument is an [`Ident`] together with a [`Type`].
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct Arg {
```

```rust
    /// The name of the argument.
    pub name: Ident,
    /// The type of the argument.
    pub ty: Type,
}

/// A block, which is a sequence of zero or more statements followed by
↪   an optional final
/// expression.
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct Block {
    /// The statements that appear in the block.
    pub stmts: Vec<Stmt>,
    /// The final expression at the end of the block.
    ///
    /// This is the value that the block resolves to when used as an
↪   expression (including as a
    /// function body). This is optional; if omitted, the block resolves
↪   to `()`.
    pub expr: Option<Expr>,
}

/// A statement is anything that performs an action.
///
/// This includes pushing to or popping from a location, but also
↪   includes semicolon-terminated
/// expressions, whose action is to resolve to the value of the
↪   expression and then discard that
/// value.
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Stmt {
    /// An expression statement is simply a semicolon-terminated
↪   expression, and it evaluates the
    /// value of that expression and then discards it.
    Expr(Expr),
    /// A let declaration is a binding of a name to a constant value in
↪   the current scope.
    Let {
        /// The name of the constant.
        name: Ident,
        /// The value of the constant.
        val: Expr,
    },
    /// A variable declaration is a declaration of a name and an initial
↪   value to store under that
    /// name in the current scope.
    Var {
        /// The name of the variable.
        name: Ident,
        /// The initial value.
        val: Expr,
    },
    /// An assignment statement is an update to the value of a variable.
    Assign {
        /// The name of the variable.
        name: Ident,
        /// The new value.
        val: Expr,
    },
    /// A push statement pushes the value of the expression `expr` to
↪   the location `loc`.
    Push {
        /// The expression to push the value of.
        expr: Expr,
        /// The location to push to.
        loc: Loc,
    },
}

/// An expression is anything that resolves to a value.
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Expr {
    /// A literal expression resolves to the value of the literal.
    Literal(Literal),
    /// An identifier expression resolves to the value named by the
↪   identifier.
    Ident(Ident),
    /// A pop expression pops a value of the given type `ty` from the
↪   location `loc`.
    Pop {
        /// The location to pop from.
        loc: Loc,
        /// The type to pop.
        ty: Type,
    },
    /// A call expression resolves to the value produced by calling the
↪   function produced by the
    /// expression `callee` with the args `args`.
    Call {
        /// An expression that resolves to the function being called.
        callee: Box<Expr>,
        /// The list of concrete location parameters provided in the
↪   call.
        ///
        /// This must contain one location for each unfilled location
↪   parameter in the callee for
        /// this to typecheck later.
        locs: Vec<Loc>,
        /// The list of arguments provided in the call.
        args: Vec<Expr>,
    },
    /// An equality expression is a binary equality operator applied to
↪   a left and right hand side
    /// expression.
    Equality(Box<Expr>, EqualityOp, Box<Expr>),
    /// A relational expression is a binary relational operator applied
↪   to a left and right hand
    /// side expression.
    Relational(Box<Expr>, RelationalOp, Box<Expr>),
    /// An arithmetic operation is a binary arithmetic operator applied
↪   to a left and right hand
    /// side expression.
    Arithmetic(Box<Expr>, ArithmeticOp, Box<Expr>),
    /// A unary expression is a unary operator applied to an expression.
    Unary(UnaryOp, Box<Expr>),
    /// An if-else expression evaluates to the value of one of two
↪   blocks depending on the value of
    /// a conditional expression.
    IfElse {
        /// The condition.
        cond: Box<Expr>,
        /// The block that is evaluated if the condition is true.
        then: Box<Block>,
        /// The block that is evaluated if the condition is false.
        else_: Box<Block>,
    },
    /// A block resolves to the value of its final expression, or unit
↪   otherwise.
    Block(Box<Block>),
}

/// An equality operator is a binary operator (an operator that appears
↪   between its two operands).
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
pub enum EqualityOp {
    /// Equality comparison (e.g. `12 == 12`).
    Eq,
    /// Inequality comparison (e.g. `7 != 3`).
    Neq,
}
```

```
impl fmt::Display for EqualityOp {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            EqualityOp::Eq => write!(f, "=="),
            EqualityOp::Neq => write!(f, "!="),
        }
    }
}

/// A relational operator is a binary operator (an operator that appears
↪   between its two operands).
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
pub enum RelationalOp {
    /// Less-than comparison (e.g. `44 < 120`).
    Lt,
    /// Greater-than comparison (e.g. `90 > 88`).
    Gt,
    /// Less-than-or-equals comparison (e.g. `32 <= 32`).
    Leq,
    /// Greater-than-or-equals comparison (e.g. `128 >= 64`).
    Geq,
}

impl fmt::Display for RelationalOp {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            RelationalOp::Lt => write!(f, "<"),
            RelationalOp::Gt => write!(f, ">"),
            RelationalOp::Leq => write!(f, "<="),
            RelationalOp::Geq => write!(f, ">="),
        }
    }
}

/// An arithmetic operator is a binary operator (an operator that
↪   appears between its two
/// operands).
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
pub enum ArithmeticOp {
    /// Addition (e.g. `43 + 26`)
    Add,
    /// Subtraction (e.g. `108 - 39`)
    Sub,
    /// Multiplication (e.g. `23 * 3`)
    Mul,
    /// Division (e.g. `414 / 6`)
    Div,
}

impl fmt::Display for ArithmeticOp {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ArithmeticOp::Add => write!(f, "+"),
            ArithmeticOp::Sub => write!(f, "-"),
            ArithmeticOp::Mul => write!(f, "*"),
            ArithmeticOp::Div => write!(f, "/"),
        }
    }
}

/// A unary operator is an operator that appears before its operand.
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
pub enum UnaryOp {
    /// A binary not operator (e.g. `!false`).
    Not,
}

impl fmt::Display for UnaryOp {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            UnaryOp::Not => write!(f, "!"),
        }
    }
}

/// An effect, representing a mutation of the value(s) in some location.
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct Effect {
    /// The location being mutated.
    pub loc: Loc,
    /// The type of the value stored in the location that is being
↪   modified.
    pub ty: Type,
}

/// A type: either unit, a named base type, or an arrow type.
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Type {
    /// The unit type; this is the type whose only value is the value
↪   [`Literal::Unit`],
    /// representing no value, the empty tuple, or an empty list of
↪   arguments.
    Unit,
    /// A base type is a single type name together with an optional
↪   location (if no location is
    /// provided, the main stack is used).
    Base {
        /// The name of the type.
        name: TypeName,
    },
    /// An arrow type is a function from one sequence of effects and
↪   types to another of each.
    Arrow {
        /// The input effects.
        eff_in: Vec<Effect>,
        /// The input types.
        ty_in: Vec<Type>,
        /// The output effects.
        eff_out: Vec<Effect>,
        /// The output type.
        ty_out: Box<Type>,
    },
}

/// An identifier.
///
/// This struct stores a [`Spur`], which is the key to an interned
↪   string stored in a
/// [`Rodeo`][lasso::Rodeo].
#[derive(Debug, Copy, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Ident(pub Spur);

/// A type name.
///
/// This struct stores a [`Spur`], which is the key to an interned
↪   string stored in a
/// [`Rodeo`][lasso::Rodeo].
#[derive(Debug, Copy, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct TypeName(pub Spur);

/// A location, such as a stack, stream, or memory cell.
///
/// This struct stores a [`Spur`], which is the key to an interned
↪   string stored in a
/// [`Rodeo`][lasso::Rodeo].
#[derive(Debug, Copy, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct Loc(pub Spur);

/// An literal's value directly appears in source code (and therefore in
↪   the AST).
```

```
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Literal {
    /// The unit value; this is the only valid value of the type
↪   [`Type::Unit`], representing no
    /// value, the empty tuple, or an empty list of arguments.
    Unit,
    /// An integer literal.
    ///
    /// This could have been created by a binary, octal, decimal or
↪   hexadecimal literal in source
    /// code.
    Int(isize),
    /// A boolean literal.
    Bool(bool),
    /// A string literal.
    String(String),
    /// A path literal.
    Path(PathBuf),
}

impl fmt::Display for Literal {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            Literal::Unit => write!(f, "()"),
            Literal::Int(val) => write!(f, "{val}"),
            Literal::Bool(val) => write!(f, "{val}"),
            Literal::String(val) => write!(f, "{val}"),
            Literal::Path(val) => write!(f, "{}", val.display()),
        }
    }
}
```