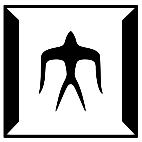

Tokyo Institute of Technology

Master Thesis

Magritte: A Language for Pipe-Based Programming

Author:
Jeanine Adkisson

Supervisor:
Prof. Hidehiko Masuhara

Student Number:
17M38029

A thesis submitted in fulfillment of the requirements
for the degree of Masters in Mathematical and Computing Science

in the

Programming Research Group
Department of Mathematical and Computing Science

August 29, 2019

Declaration of Authorship

I, Jeanine Adkisson, declare that this thesis titled, "Magritte: A Language for Pipe-Based Programming" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"*This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.* "

Doug McIlroy

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Masters in Mathematical and Computing Science

Magritte: A Language for Pipe-Based Programming

by Jeanine Adkisson

This work proposes *Magritte*, a general-purpose language that is viable as a shell scripting language. Like shells, it manages concurrent processes which are composed by connecting implicit input and output streams with a pipe. Unlike traditional byte-stream-based pipes, however, *Magritte* pipes can process rich values, enabling their use as a more general composition mechanism, which enables Magritte to express concurrent processes as normal functions.

Included in the design are modern language features such as data structures and lambda functions with lexical scope, as well as systems for automatic process cleanup and error handling. The syntax is also designed to optimize for command-line usability.

This work also presents an implementation, along with a proposed method of integration with a POSIX system.

Acknowledgements

The Magritte interpreter and the Miller-Rabin example program were a joint work with Johannes Westlund.

I would also like to thank my advisor, Prof. Masuhara, for encouraging and supporting me in this work.

Contents

Chapter 1

Motivation

The UNIX shell programming model has played an important role in integrating applications and operating systems by composing programs—spawning independent programs in parallel to communicate over operating-system pipes. Beyond the most simple tasks, however, `bash` and similar tools break down, due to various language deficiencies.

1.1 Pipe-Based Languages

In this paper, we use the term **pipe-based language** to mean a programming language intended to be used on a command line, with the ability to spawn many processes which communicate through synchronous channels or pipes in an ad-hoc manner—and in which this facility is the primary method of composing different functions or units.

Our overarching goal is to create a language and a programming system that retains the pipe-based programming and interaction model of `bash`, but allows for large programs in a way that existing shell languages do not.

1.2 General Purpose Languages

In this work, we will use the term **general-purpose languages**, to refer to languages such as C, Python, or Java, which provide data structures and abstraction tools, and run on an operating system and can access standard APIs.

Because they have the proper abstraction tools, they can be used to write **large programs**—programs that are difficult to understand by one person in their entirety, because of the fundamental complexity of the problem space. This is in contrast to small scripts, which may have high incidental complexity, but are fundamentally accomplishing a small task.

1.3 The Shell-Language Design Problem

The **shell-language design problem** is a well-studied problem in programming language design: how can we make shell languages have more useful semantics, and be more appropriate for larger programs? Many attempts have been made to solve this problem (discussed in Chapter 2), and many works-in-progress[1], are being designed at the time of publication. We hope that this research contributes meaningfully to this effort.

[1] At the time of publication, the maintainers of the Oil project [*Chu, 2019*] maintain a list of such projects at `https://github.com/oilshell/oil/wiki/ExternalResources`.

1.4 The Desktop-Scripting Problem

Related to the shell-language design problem is the **desktop-scripting problem**: How should unrelated programs written in different languages be integrated—especially in an ad-hoc manner in a desktop environment? Such a task can require a large amount of glue code, written by users who are unfamiliar with the inner workings of the programs they are using. Notable attempts at solving the desktop-scripting problem include the TCL language [*Ousterhout, 1989*] and Guile Scheme [*Blandy, 1998*]. However, while most of these approaches use a large, robust language, it still remains difficult to integrate them with external programs—instead putting the burden on those programs to integrate with their system. We believe a shell-based approach is promising, because if we are successful, our language could use the already-ubiquitous UNIX calling convention to integrate unrelated programs.

1.5 Evaluation Criteria

In order to evaluate a solution to the above problems, then, a new language would have to meet at least the following criteria:

- It is a viable general-purpose language: i.e. it is appropriate *for writing large programs.*
- It is a viable solution to the desktop-scripting problem: it is appropriate *for scripting unrelated programs.*
- It is a viable solution to the shell-language design problem: it is usable *as an interactive system shell.*

We take the above as our main research goals.

1.6 Overview

We will first introduce the current state of the art and related work in Chapter 2, including existing shell languages, as well as previous attempts to create shell-like general-purpose languages or extend shell languages with general-purpose features.

In Chapter 3, we will describe the general design requirements for a language in the intersection of pipe-based and general-purpose languages. This will inform and justify our approach to the design of Magritte.

We will describe the design of Magritte in Chapter 4, along with several design challenges and choices. Then we will describe in detail the reference implementation in Chapter 5, and elaborate on several implementation challenges and our solutions.

In Chapter 6, we will evaluate Magritte based on the above criteria, and we will argue in Chapter 7 that Magritte represents a viable direction for solving the shell language design problem.

Chapter 2

Background

Many attempts have been made at the shell design problem (Section 1.3), to improve the design of shell languages without sacrificing the usability and composition model of the underlying language. In this section we will outline some of the problems with traditional shell languages that these languages attempt to solve, the strategies each takes, and which problems remain unsolved.

2.1 Traditional Shell Languages

The most popular family of shell languages are based on the Bourne Shell (hereafter `sh`), and its modern variant, the Bourne Again Shell (hereafter `bash`). This family of languages, which we will refer to as **traditional shell languages**, or **shells** for short, include `zsh`, `fish`, `csh`, and others. Traditional shell languages vary in specific features and syntax, but all contain a large core semantics that influence the way programs are written.

The core unit of computation in a traditional shell language is a **command**. Commands are intended to be used the same way in scripts and through the interactive REPL[1] system, and can refer either to external programs written in other languages, builtin commands from the shell language, or user-defined functions in a script.

Shell builtins and user-defined functions often have capabilities that external programs do not, but the basic calling convention remains the same: a command receives a list of byte vectors, referred to as an **argument vector** or `argv` for short, and performs computation with full access to the operating system. Additionally, a command will be provided with three implicit file descriptors: A readable descriptor called the **standard input** or `stdin`, and two writable file descriptors called the **standard output** (or `stdout`) and the **standard error** (or `stderr`). Rather than returning a value, as in most programming languages, a command returns an **error code** (normally `0` to indicate no error), and provides all output by writing data to `stdout`.

This convention enables commands to be connected via **pipes**: a special operator that connects the `stdout` of one command to the `stdin` of another. Because reading and writing from file descriptors is a blocking, ad-hoc operation, the pipe operator is able to run both commands in parallel, such that both sides use read and write operations to synchronize and communicate. In practice, though, synchronization can be difficult because the pipe provides a **buffer** so that the writing side can continue without the data being read.

Variables in traditional shell languages are either global or **dynamic**—they are scoped to a particular stack frame, and visible from all lower stack frames. They are

[1] Short for **r**ead-**e**val-**p**rint-**l**oop, a REPL is another word for a console interface: in which a segment of user input is read, then evaluated, and the result printed to the console, in a loop.

optionally passed to external programs as **environment variables**, which visible from an operating system API available to most programming languages. In `bash`, variables from parent stack frames are also **mutable**, as long as the stack frame is in the same thread.

This interface has been the common denominator of programs for many decades[*Mahoney et al., 1989*], and serves as the most common way for end users to integrate unrelated OS programs in an ad-hoc manner.

2.2 Unsuitability for Large Programs

We argue that traditional shell languages are unsuitable for large programs, because they lack necessary features.

2.2.1 Lack of Usable Data Structures

A priori in a shell language, the only possible value type is a plain sequence of bytes. Numbers, processes, and files all share the same representation as strings. Since this extremely limits the expressiveness of the language, many traditional shells, including bash itself, have attempted to add arrays and associative arrays as standard objects. However, these are not generally treated as first class values: they are not allowed to be passed into or returned from functions, or importantly, passed through pipes.

These values are therefore relegated to a second class of value, one for which pipe-based composition is not possible. Furthermore, serialization through pipes is generally not possible except in a single-threaded case, due to the undifferentiable nature of byte pipes (Section 2.2.2).

It is also very difficult in shells to capture return values from functions, either as strings or as simple values. For example:

```
f() { echo a; echo "b c"; }

# assignment captures the whole string
y=$(f)
echo "$y" # => writes the string "a\nb c"

# iteration breaks on whitespace
for c in $(f); do echo "$c"; done # => 3 lines: a, b, c

# read-loop spawns a new isolated subshell
f | while read line; do echo "$line"; done
```

Because one of the values that `f` outputs contains whitespace, it is considered to be a separate value for iteration. We can preserve the entire string, but none of these separate the output properly into the two writes "`a`" and "`b c`". One way around this is to use a **read loop**, which depends on the behavior of the builtin `read` to properly split lines[2]. However, this will corrupt the data in a similar way as soon as the data contains a newline.

In order to return rich values from functions then, a strategy that is sometimes used is to set a global variable:

[2]Technically, this should use the `-n` flag to disable backslash handling, plus a few more special-purpose flags to turn off some convenience features that can corrupt the input.

```
array-one-two() { RET=(1 2) ;}

array-one-two
echo "${RET[1]}" # => 2
```

However, arrays are not generally copyable to other variables, which means it is not easy to bind a variable to the current array-value of `$RET`. The inability to return or accept these kinds of variables as arguments means that these structures are not viable as data structures.

2.2.2 Undifferentiable Byte Pipes

While pipes provide a convenient way to chain together programs, anything more than the simplest data can be corrupted through interleaving. Even if that were not the case, there is no reliable way to separate values that come through a pipe.

To illustrate, consider the fairly common architecture of producer/consumer: A *producer* process produces values that are processed in parallel by multiple *consumer* processes, and the values are collected in a single output. This might be expressed in bash as:

```
# *consume* the stream, labeling every line
label() { while read x; do echo "$1$x"; done ;}

# process the stream with two threads
split() { label a & label b & ;}

# *produce* and process the numbers 1-100,
# limiting output to the first 3 lines
seq 100 | split | head -3
```

With this code, a user might expect the output to be three lines, each consisting of a letter `a` or `b`, and a number, for example:

```
a1
a2
a3
```

```
b1
a3
b2
```

```
a2
b1
a3
```

Unfortunately, when we run this process, the outputs from the `label` function become interleaved[3], resulting in outputs such as:

```
a2
b1
b
```

```
b12
a
a4
```

```
a
b12
a34
```

This behavior is in accordance with the Linux User's Manual [*Linux man-pages Project, 2018*]:

> The communication channel provided by a pipe is a *byte stream*: there is no concept of message boundaries.

[3] We have observed some behavior in the output that cannot be explained simply by interleaving, suggesting there may be some other race conditions in play.

2.2.3 Lack of Modularity

In writing large programs, it is desirable to write modular interfaces that bundle data and code, with public and private members.

This is generally impossible in traditional shell languages, as all functions share a single global scope, and variables must be either global or stack-based. The closest approximation would be using a name-mangling strategy (using "`:`" as a normal identifier character):

```
my-module::public-member() { ... ;}
my-module::__private-member() { ... ;}
```

Furthermore, all variables are stored on the stack, and there is no support for nested or mutable data structures. Therefore shared data must be global-only, and the programmer must use manual name-prefixing to avoid clashes with other modules. This precludes any kind of object-like semantics.

2.2.4 Global-Only Exception Handling

It is well-known that exception handling in `bash` is difficult and error-prone. The only way to react to an error is to use the builtin `trap` command, which can register a global handler for the `ERR` event. However, it is not possible to control the extent of this exception (it always unwinds the stack globally), and only one handler can be registered at a time. Accordingly, programmers must be very careful lest they unintentionally clear an exception handler by registering a new one.

2.3 Related Work

The following languages have attempted to extend the notion of a shell to have more modern features. We argue, however, that they do not constitute a complete solution to our research goal.

2.3.1 Rc Shell

Rc shell [*Duff, 1990*] was planned as the default system shell for the Plan 9 operating system from Bell Labs. Rc Shell fixes many whitespace inconsistencies from bash by representing every value as a *list* of strings—plain strings being simply a list of length 1. However, these lists fall short of being usable as data structures, as it is not possible to nest them. According to the manual:

```
Argument lists have no recursive structure, although their syntax
may suggest it.  The following are entirely equivalent:
     echo hi there everybody
     ((echo) (hi there) everybody)
```

Rc Shell offers no other data structure primitives. Its error handling remains global-only, calling a user-defined global function when an operating system signal is received. Additionally, Rc shell does not change the semantics of pipes in any significant way—they remain byte-based, and undifferentiable.

2.3.2 Es Shell

Es Shell [*Haahr and Rakitzis, 1993*] is an extension of Rc shell that adds lambda functions using the syntax `@ arg { body }`. This is accomplished by serializing functions as strings, together with their closure:

```
$ let (x = 1) { y = @z { echo $x; echo $z }}
$ echo $y
@z %closure(x=1){%seq {echo $x} {echo $z}}
```

In order to capture return values, Es introduces a special syntax `<={ command }`[4] to capture the return value from a command. Return values are produced using the `return` keyword, and are integrated with the error code return system for external programs.

Function output, however, remains strictly byte-based, as do all of the channels, so pipe-based composition is not possible.

2.3.3 Scheme shell (scsh)

Scheme shell, or `scsh`, [*Shivers, 1994*] is a shell-programming environment that uses scheme syntax. It mixes external program semantics with scheme semantics, essentially using scheme as a metaprogramming layer over calls to external programs and low-level builtins. While this enables high-level programming in scheme, we find that, with the exception of the shell layer, scheme syntax lacks sufficient linear composition (see Section 3.2.1) to be used effectively as an interactive shell. Additionally, there is a very large semantic gap between the scheme semantics used for in-language constructs and the semantics of external programs, which means that any kind of interfacing between the two that goes beyond simple orchestration or unquoting requires a much more complex interface than simple pipes.

2.3.4 Xonsh

Xonsh [*Scopatz, 2018*] is a syntax extension of Python that allows shell behavior. Being a syntax extension, it needs to determine the difference between "subprocess mode", in which the syntax behaves in a shell-like manner, and "python mode", in which case the code is parsed as normal Python. The translation is not direct: Python functions cannot be directly used in subprocess mode without manual registration, and a series of special embed syntaxes are necessary to call subprocesses from Python mode in different ways. Additionally, the gap between Python semantics and shell semantics is large enough that, like scheme shell, deep integration is difficult.

2.3.5 Windows Powershell

Windows Powershell has all but replaced the old `cmd` shell for modern Windows programs. It is an object shell—in that pipes consume objects as well as bytes. It is also completely integrated with the `.NET` API that runs most programs on modern Windows systems.

Because Windows APIs are unified into a single rich set of functions, this means that Powershell can integrate unrelated Windows programs in different languages, so long as those languages use the `.NET` API. This effectively solves the desktop-scripting

[4] In the paper, this syntax is described as `@<...>`, but the current implementation uses the syntax described here.

problem for Windows environments. However, though Powershell can be run outside of Windows, this advantage is mostly lost in other environments where integration with `.NET` is not so ubiquitous.

Additionally, while the language does support some object-oriented features, it is still not widely considered to be appropriate for larger programs. And while it does support anonymous lambda functions, it does not support pattern-matching over nested structures, so most functional patterns are not supported. Finally, naming conventions of the standard library are unnecessarily heavy, requiring long names and parameters such as `ValueFromPipelineByPropertyName` for accessing basic language functionality.

2.3.6 Elvish shell

Elvish shell [*Xiao, 2019*] is a new shell, developed concurrently with Magritte, with value-based semantics. It allows for nestable list and map data structures, as well as lambda functions with closure. It contains both value and byte-based pipes, allowing for a large amount of pipe-based programming.

A major difference between Elvish and Magritte is that Elvish does not allow for ad-hoc reads from value channels. All functions available to read from the standard input consume the entirety of the input in one loop. It is also not possible to refer to channels explicitly as values, or pass channels over channels. In fact, channels are only available implicitly through the pipe operator. It could be argued that this is sufficient for shell programming, but it does restrict the concurrency system somewhat, such that for example the Magritte program listed in Section A.2 would be impossible in Elvish.

Chapter 3

Design Constraints

In this section we discuss several feature requirements and design considerations for a pipe-based language to be viable for large programs.

3.1 Programming With Values

3.1.1 Value Pipes

It is desirable in a general purpose language to be able to create and use complex data structures, and to use them freely throughout the language. In particular, if we are to allow for pipe-based composition to be the core composition method for large programs, we must address the problems with pipes discussed in Section 2.2.2 and Section 2.2.1. To do this, we must ensure that channels are *consistent*—that is, that every read corresponds to exactly one write.

We propose to accomplish this by using **value pipes**: a strategy wherein a value is the smallest atomic unit of communication, and we allow rich values to be passed wholesale through pipes and emerge exactly once and intact on the other side.

3.1.2 Capture and Substitution

For processes that output values, we need to support a mechanism to capture those values for use in variables, data structures, and function arguments, similar to backticks or `$(...)` in bash. This enables processes to be used like functions, which return data to their caller.

But we showed in Section 2.2.1 and Section 2.2.2 that in most shells, rich values are not writable to output streams, nor is it possible to consistently separate values written to byte streams, and thus `stdout` cannot effectively be used as a return path for values. Es Shell accounts for this by introducing a *return value* which is separate from stream output and can be accessed via special-purpose call syntax.

If we allow values to be written to output streams, however, we can directly capture and separate values written to the output.

3.1.3 Modern Language Features

Users will expect a modern programming language to have, at minimum:

- *Lambda functions with closure.* This requires the introduction of lexically scoped variables.
- *Dynamic variables.* Most shell languages already include these, as OS Environment variables are dynamic by nature.

- *Product structures with support for open recursion.*[1] A prototype-based object system is sufficient for this.
- *Sum structures.* In an untyped language, a method for pattern matching over nestable heterogenous lists is sufficient.

For large programs, an object system is desirable as a way to provide user-defined types and abstractions. The existing foundations of shell languages already make use of the UNIX *environments* API, which consists of a map of strings to strings, together with a parent pointer. Therefore, if we allow environments as first-class values, they could be used in a straightforward manner as prototype-based objects.

For programs that deal with syntax or other kinds of trees, it is convenient to have compact nestable data structures that support pattern matching, which can be made possible with nestable vectors.

3.2 Syntax Design

In this section we will outline the goals and requirements of a shell syntax.

3.2.1 Linear Composition: Why Not Lisp?

A common choice for "simple" syntax in programming languages is to use a variant of Lisp, in which all expressions use parentheses for delimiting subexpressions, and no infix operators are allowed. This lack of infix operators has led some to claim that Lisp "has no syntax" [*Graham, 2002*]. However, when we evaluate the performance of Lisp syntax in a REPL context, we immediately find that they lack a syntax property that we call *linear composition*. To illustrate, consider an example in which a user has typed some code that they wish to execute (where ▮ represents the user's cursor):

```
~> (fan 4 fetch-webpage urls)▮
```

In this example, the user intention is to retreive a list of urls from a file, and fetch them from a network using 4 threads. Now suppose the user wants to extract the titles from these pages, as well as sort them. The user would need to return the cursor to the beginning of the line (by pressing arrow keys or a macro such as `^A`), add the required code, and then return to the end to insert the final parentheses.

```
# move cursor to the beginning
~> ▮(fan 4 fetch-webpage urls)

# insert new code
~> (sort (map get-title▮(fan 4 fetch-webpage urls)

# move to the end
~> (sort (map get-title (fan 4 fetch-webpage urls)▮
```

[1] We use the terms **sum structures** and **product structures** to refer to what would normally be called "sum types" and "product types". In an untyped language context, we choose to emphasize the shape of the data structure itself over its type.

```
# add parentheses
~> (sort (map get-title (fan 4 fetch-webpage urls)))
```

In this example, it was necessary to create a deeply nested expression to express a task that is essentially a sequence of sub-tasks. Worse, the sub-tasks are actually listed in reverse order from the flow of data, meaning that any additional component will require expansion to the left. Compare to an example using Magritte syntax:

```
~> fan 4 fetch-webpage urls | each get-title | sort
```

This expression not only requires no nesting, but also allows extension by adding components directly to the right. This is an important syntax property we call **linear composition**: the ability to use previous results by inserting more text to the right, without backtracking with the cursor.

To be clear: the problem with the above example is *not* parentheses, though they contribute to the overall nesting of the syntax. Rather, the problem is the prefix-only calling convention that requires a back-and-forth movenent of the cursor, and causes the expression to expand leftwards, opposite of the direction of the user's typing.

While some languages have some support for linear composition using operators such as Clojure's arrow macros or OCaml's `|>` operator, the integration of these macros with the underlying subsystem is shallow, and most tasks require a large amount of such nested code.

3.2.2 REPL Flexibility: One-Dimensional Code

A user typing on a REPL interface will often find it difficult and cumbersome to manage whitespace correctly. This is because code in a REPL is generally contained in one semantic "line", even if the text wraps. We call this **one-dimensional code**, in contrast with **two-dimensional code**, which can be edited in a text editor, and has room to be formatted in a neat manner.

Because of the one-dimensional property of REPL input, it is necessary for a syntax targetting REPL interfaces to be whitespace-agnostic, or at least to preserve the property that any expression usable in a file is also expressible in a single line. This property necessarily precludes indentation-sensitivity.

At the same, users expect a shell to not require any special character (such as "`;`") to end a command or section—to simply enter a newline (or press enter at a prompt) to separate commands. This can introduce some parsing ambiguities when combined with pattern-matching. We describe our approach to solving these ambiguities in Section 4.1.4.

3.2.3 Shell Flexibility: Barewords

In a shell, it is necessary to be able to call external programs without quotation. In most languages, unadorned strings are interpreted as variables or keywords. Instead, in shells, these are interpreted as strings. Therefore, code such as:

```
git "commit" "-m" "hello world"
```

does not require quotations as long as the argument does not contain a space or other terminating character:

```
git commit -m "hello world"
```

Accordingly, any variables and keywords we introduce into the language must be marked with special syntax or sigils to distinguish them from barewords.

3.2.4 Macros

While Magritte does not yet support macros, a language operating in a very dynamic and interactive environment is likely to require syntactic abstraction features. Thus it is desirable to design the syntax structure to be easy to generate and manipulate in the way that Lisp syntax is. We describe our approach to solving this problem, inspired by the Dylan language, in Section 5.2.1.

3.3 Automatic Process Cleanup

3.3.1 Interruption

In shell programming, we tend to compose infinitely running processes together as pipeline elements. Thus we require a well-defined semantics of process **interruption**[2]: automatic clean-up of processes that will no longer be used. Consider the following Magritte code:

```
read-lines tmp/large-file                              (1)
  | take 10                                            (2)
  | each (?line => do-expensive-work $line)            (3)
```

In this example, three processes are spawned concurrently: (1) a process with an open file that writes one line at a time to its output, (2) a process that reads 10 times from its input, writes each entry to the output, and then exits, and (3) a process that reads every input and calls a function to perform an expensive task. A user's intent when typing such code may be to read 10 lines from a file and synchronously perform an action on each line.

A user will also expect that, after the first 10 lines are processed and the `take` function returns, the file will be closed, and all three processes will exit. This expectation is despite the fact that process (1) is specified to read the entire file, and process (3) is an infinite loop.

In a naïve implementation using synchronous channels, process (1) will never be able to write more than 10 lines, and will remain blocked on its output with the file open forever. Similarly, the call to the **each** function will never be notified that its input has finished, and will block forever on its input stream.

3.3.2 Compensation

In interacting with an operating system, it is necessary to manage side effects, and gracefully recover or restore state in the case of an interruption. Such an error-handling system would also be a way for user code to directly observe interruption.

Without compensation of errors, the only way of observing forever-blocked processes would be to inspect the process table, or to observe the memory footprint of the program. Thus, we can define the primary task of the interruption system as *running a process's compensation actions at the appropriate time.*

[2] We use the term **interruption** in the generic sense—to mean the halting of normal flow in a process, due to some external event. It is unrelated to hardware or OS signalling.

3.3.3 Lazy Interruption vs. Eager Interruption

In a system such as this, there is a language design choice that must be made—whether interruption is **eager**, in which processes are interrupted immediately upon closing a channel, or **lazy**, in which processes are only interrupted upon interaction with a closed channel. Both approaches have advantages and drawbacks. In making this choice, we desire that the behavior of interruption be *predictable*—however, there are two competing viewpoints for predictability.

Consider the following example, with the assumption that `do-other-operation` does not write to the standard output:

```
(produce-values) = (
  put 1 2 3
  do-other-operation
)

produce-values | take 3
```

The `produce-values` function will write three times to the pipe, and then continue to do other processing in-thread that does not write to the standard output. The final `take 3` operation will return after 3 inputs are read. It is important to decide, then, whether the process on the left should be interrupted in the middle of `do-other-operation` (eager interruption), or whether it should be left alone until it attempts to write a value (lazy interruption).

From the perspective of someone spawning a process, eager interruption can seem more predictable, as they can guarantee a point at which the process has stopped doing work.

However, from the perspective of a function author, lazy interruption is more predictable, because the author can identify precisely which points in the code have the potential to be interrupted—those points which run a **`put`** or a **`get`**. Contrast this with eager-interruption semantics, where any point in the code may be interrupted, introducing the need for users to mark critical sections and be very careful with implementing stateful algorithms.

On the other hand, lazy interruption has the disadvantage that a producing process must do enough work to produce one more value than will be consumed. If each value is relatively cheap to produce, this is not a problem, but in the case that the values are expensive to produce, this would result in a large amount of unnecessary work.

3.3.4 Closing of Channels

Given that multiple processes may be reading from and writing to a channel, it is often the case that a communicating process will end when there are other processes still communicating over the channel. It would not be appropriate in this case to interrupt other processes attached to the channel, as they are still able to communicate.

We define our guiding principle for the appropriate time to interrupt a process as: *A process is interrupted exactly when it can no longer be woken up.* When a process is blocked on a synchronous channel, it will be woken up as soon as another process communicates on the other end. Therefore the appropriate time for it to be interrupted is when it is blocked on a channel that will never receive any more operations.

This can be difficult to detect when a reference to a channel can be passed anywhere in the program as a standard value. Luckily, the arrangement of pipes and channels are usually specified at process spawn time. We can therefore relax our constraint to guarantee that channels will be closed at appropriate times in *common architectures*, and that they never close if there are active readers or writers.

3.3.5 Reopening Channels

Some systems, like UNIX named pipes, allow a channel to be re-used by new processes after it has been closed [*Linux man-pages Project, 2018*]. This is, however, not a desirable feature, as it can lead to some unexpected races between a channel closing and a new process spawning. Consider the example:

```
c = (make-channel)      # create a new channel
& count-forever > $c    # write infinitely
& take 10 < $c          # read 10 elements and exit
put 10 > $c             # write once from a new process
```

This example represents an unavoidable race with first-class channels: between `take 10` closing the channel and **`put 10`** opening the channel for writing. If we allow channel reopening, we will either block forever, or insert the number 10 into the stream, depending on which happens first. However if we do not allow channel reopening, we can say that, if a process initiates a read or write on a closed channel, it is immediately interrupted. In this case, both sides of the former race have the same termination behavior—the process is interrupted when `take 10` returns.

3.3.6 Tail Call Detection

In order for channel closing to work properly, it is necessary to identify tail calls that change the channel environment. For example:

```
(
  process-data
  write-log > $c1
) > $c2
```

In this example, the tail-position call to `write-log` is redirected, and `$c2` falls out of scope. In this case, it is important that `$c2` is deregistered before entering the tail-call, to allow other processes to continue.

3.3.7 Masking Interruptions

We must include a facility to control the extent of interruptions. This is because an interruption semantics can make it difficult to perform final calculations after a channel is closed.

At the most basic level, we define the extent of channel-closing interruption as unrolling the stack until the channel is no longer in the current frame's corresponding input or output. However, more manual control of the extent is still needed. Consider the following example, which sums all numbers from the standard input:

```
(sum) = (
  total = 0
  each (?x => %total = (add %total %x))
  put %total
)
```

In this example, the function **each** will consume the input stream and mutate a lexical variable (marked with `%`). After the entire input stream is consumed, we wish to output the resulting `%total` value.

However, with the semantics described above, the **each** function will loop until it receives an interrupt signal from the input channel closing. Since containing scope still has the same channel set as its input, we will unroll past the point where we can execute the final "**put** `%total`".

Thus, in order to continue properly after consuming a stream, a mechanism to capture this unrolling is required.

3.4 UNIX Integration

For a shell language to be viable for script-writing or interactive use, it must have robust OS integration. We focus on the UNIX model in particular. In order to be usable as a shell, it is desirable to unify the notions of an in-language function call and an external program call as much as possible. For this reason we say that our goal is to be able to transparently replace Magritte function calls with external program calls, in the case that they are equivalent. For example, a call to `filter` that only searches for certain strings should behave the same and have the same syntax as a call to `/usr/bin/grep`. This is a feature of every successful shell—while in-language functions may have extra features, they fundamentally behave the same as external programs.

This is a key difference between Magritte and projects such as `scsh` and `xonsh`: Magritte's computation model is similar enough to the shell that this replacement property should be possible in many cases.

3.4.1 Marshalling

In order to call external functions with rich-valued arguments, we introduce the need for **marshalling**: since external programs can only receive byte vectors as arguments, we need a flexible and user-configurable way of rendering and parsing values to and from strings.

3.4.2 Marshalling Over Pipes

Additionally, the language must provide a compatibility layer between UNIX byte pipes and value pipes. This is extra challenging because external processes require values to be separated in a wide variety of ways: spaces, newlines, tabs, or even null bytes—and they also often support different ways of escaping content. It is necessary, therefore, to provide a way of specifying different rendering and parsing behavior for each command. Worse, though, many programs exhibit different parsing behavior when passed different flag arguments, so it is necessary to either specify pattern-matches against full command vectors, or ask the user to manually parse or render strings through a separate command.

Chapter 4

Description of Magritte

We propose a language that meets the requirements of Chapter 3.

4.1 Syntax Overview

Here we will briefly describe each syntax class, so that the following examples are more readable. See Appendix B for a more complete and formal description of the syntax.

We will include an imprecise EBNF approximation of the syntax for readers who are more comfortable with this kind of specification, but the syntax is more precisely defined using a strategy described in Section 5.2.1.

4.1.1 Token Syntax: Barewords and Non-Reserved Symbols

The syntax of Magritte includes barewords, as discussed in Section 3.2.3. Additionally, we do not reserve any keywords - every unadorned string is a bareword in Magritte. Language features are expressed in terms of punctuation designed to be intuitive to those familiar with shell languages, and all unadorned words are either command names or barewords. All command names come from the same namespace as variables, and are not treated specially based on their names. We do have a small number of builtin functions, but their semantics remains much the same as user-defined functions.

Because they are often used in filenames and expected as arguments of commands, we also do not reserve the symbols "`.`", "`/`", "`:`", "`,`", or "`-`", treating them the same as other characters in barewords. Because it may be used in file-globs in the future, we reserve but do not use "`*`". This restriction means that we must be very economical with respect to choices of features and use of syntax.

4.1.2 Grouping & Lines

As with many shell languages, parsing of Magritte code begins by splitting the code into semantic **lines**:

```
group ::= line*
```

These lines are more complex than simple newline-separated fragments, however. They can be continued, contain nested forms, or be specified inline using semicolons. The definition of what constitutes a line is described precisely in Appendix B, but for the purposes of this chapter it is sufficient to think of a line as being a grouping that is ended by either a newline or semicolon (`;`), is continued by infix tokens such as "`=>`", "`=`", and "`|`", and is nestable—newlines within parentheses or other nesting

forms do not end the containing line, but their contents may be split into lines as well.

4.1.3 Line Syntax

Each line is broken up with various infix operators, indicating control flow, compensation, and more. We will explain each of these in turn.

```
line ::= assignment | spawn | instr
spawn ::= "&" line
assignment ::= lhs+ "=" expr+
lhs ::= access (* see Data Structures *) | BARE
instr ::= cond ("%%" | "%%!") cond | cond
cond ::= pipe ("&&" | "||") (pipe "!!" cond | cond) | pipe
pipe ::= pipe "|" with | with
with ::= command redir+ | command
redir ::= (">" | "<") expr
command ::= block | expr+
block ::= "(" group ")"
```

A line can contain either an assignment, a spawn command, or an instruction to run with side-effects (e.g. a command).

We will quickly summarize the line-syntax categories from the outside to the inside. The loosest binding operators of a line are the optional **compensation** operators `%%` and `%%!` (Section 3.3.2). The next level is conditional expressions, indicated by `&&` and `||`, with an optional else parameter: for example "`cond && if-true !! if-false`" or "`cond || if-false !! if-true`". Following are pipes that can chain commands together in a pipeline (`a | b | c`), followed by specific channel redirection operators "`<`" and "`>`". Finally, a command can be either a parenthesized group (a **block**) or a command vector of expressions.

4.1.4 Expression Syntax

Unlike the line syntax, which is intended to represent side effects and commands, expressions are intended to produce one or more values to be used as arguments, values to be assigned, or elements of other expressions. We will describe each in turn.

Variables & Constants

Variables come in two forms: **lexical** `%x` and **dynamic** `$x` (See Section 4.2.1). Strings are either delimited with quotes or left bare. For example, `magritte`, `"magritte"`, and `'magritte'` are all equivalent.

Numbers are written in the usual fashion. We do not currently distinguish between floats and integers, so `1.0` and `1` are considered equivalent.

Substitution

A **substitution** is a type of expression consisting of a group surrounded by parentheses:

```
expr ::= ... | subst | ...
subst ::= "(" group ")"
```

However, this does introduce an ambiguity with the previous definition of `command`—a group surrounded by parentheses could be interpreted either as a block or as a command with a single `subst` expression. In this specific case we say that the block syntax takes precedence, and commands cannot consist of a single substitution. Should a user wish to execute a single substitution as a command, the `exec` builtin function is provided.

Data Structures

Square brackets delimit **vectors**, which can be nested. For example, the expression "`[node [leaf 1] [leaf 2]]`" is a vector containing the string `"node"` and two vectors `[leaf 1]` and `[leaf 2]`. Square brackets are defined as `free_nl` (See Section B.2.2), which means that newlines are ignored at the top-level, and substitution must be performed using parentheses.

Curly braces in argument position define **environments**, which are used in Magritte as prototype objects (See Section 4.2.4). The inside of curly braces are treated as a group, in which all assignment syntaxes are available. For example:

```
my-env = {
  a = 1
  b = [$a 1]
}
```

Environments can be explicitly accessed with "`!`" syntax: `$obj!key`. This token was chosen because it is already reserved by `bash`, so users are accustomed to needing quotation marks to use it as a string, unlike "`/`" and "`.`", which are commonly used in filenames (see Section 4.1.1).

Functions & Patterns

Lambda function syntax is similar to:

```
lambda ::= "(" (pattern+ "=>" line*)+ ")"
```

These are also defined using parentheses, but the special token `=>` is used to separate the arguments from the body. The left-hand side of the arrow is a **pattern**, and the right-hand side is parsed as a group of lines. For example:

```
add-one = (?x => add 1 $x)
```

The syntax `?x` represents a type of pattern called a **binder**, which introduces a new variable assigned to the matched value.

Multiple clauses and nested patterns are also possible. For example, this function outputs all the leaf values in a binary tree:

```
iter-tree = (
  [node ?l ?r] => iter-tree $l
                  iter-tree $r
  [leaf ?v] => put $v
)
```

This function has two **clauses**: The first can only be run if the argument is a three-element vector beginning with `"node"`; if so, it will run two recursive commands with the matched elements. The second can only be run on two-element vectors beginning with `"leaf"`; if so, it will output the matched value.

Note that there is no need for delimiters around or between clauses, nor is the indentation strictly necessary: a new clause is defined to begin at the beginning of a "line" that contains a `=>` token (See Section 5.2.1 for a detailed explanation of how this is accomplished).

For named functions with only one clause, we also support a syntax sugar for function definition:

```
(foo ?x) = (
  y $x
  z $x
)

# equivalent to
foo = (?x => y $x; z $x)
```

Any clause or vector pattern can use the special syntax `(?x)` (a "rest-pattern") to match any number of values and bind a vector. For example,

```
# a function with a variable number of arguments
(f (?args)) = ...

# a function that returns the head and tail of a vector
(shift [?head (?rest)]) = put $head $rest
```

These are currently only allowed as the final pattern.

Because we use different syntax for binders and variables, we are able to allow matching values against the content of a variable, which is often a tricky problem in pattern-matching implementations. For example:

```
(extract-tag ?t ?x) = (
  [$t ?v] => put $v
  ?other => put $other
) $x
```

The pattern `[$t ?v]` will only successfully match a vector whose first element is equal to the value of `$t`. The second element, however, can be any value, and will be bound to the binder `?v`.

4.2 Values and Variables

In this section we will describe the basic value semantics of Magritte.

4.2.1 Lexical vs. Dynamic Variables

In order to support both lexical and dynamic variables, we introduce a separate syntax for lexically scoped variables at variable reference and mutation points, using a `%` instead of `$` (e.g., `%x`).

Assignment (`x = 1`) will bind a variable both lexically and dynamically, using a plain syntax which allows us to maintain compatibility with standard environment files.

Both lexical and dynamic variables behave dynamically at lookup time, but free lexical variables are captured and included at the top of the scope at function call time. For example,

```
# dynamic $x, captures nothing
(g) = (put $x)
(f) = (x = 1; g)
(f) # => 1

# lexical %x, is saved from the local environment,
# and supersedes the dynamic binding.
g = (x = 2; (=> put %x))
(f) = (x = 1; g)
(f) # => 2
```

For mutation, we use assignment syntax, but with the location (dynamic or lexical) specified on the left-hand side, e.g. `%x = 1` or `$x = 1`. These expressions will raise a compile-time error or a runtime error, respectively, if the variable is not bound. By separating binding syntax from mutation syntax, we are able to statically determine every lexical variable's scope with no need for declaration.

4.2.2 Lambda Functions

Function values have an attached **closure environment**, which copies references to any free lexical variables at the time of creation. This environment is spliced in to the top of the running environment at function-call time. It is possible to mutate these references, and the mutation will be visible to other functions (see Section 5.1.2). Function clauses are tried in order, and only the first successfully matched clause is evaluated. In the case that no clause matches, the process will crash.

4.2.3 Nestable Vectors

Vectors are a straightforward extension of `argv` vectors: an immutable ordered collection of values, which can be nested. A common pattern is to use vectors that have strings at the beginning to represent data variants: for example, nodes of a tree:

```
a-tree = [node [node [leaf 1] [leaf 2]] [leaf 3]]
```

These can be matched in lambda arguments by patterns such as `[node ?x]`. A typical strategy for traversing this kind of structure might be a function that outputs all leaf node values to its output in a predefined order, to be consumed by another process.

The builtin function **`for`** takes a vector as an argument and outputs each element in order, so that a vector can be traversed with:

```
for [1 2 3 4 5] | each (?el => ...)
```

Vectors can also be called as functions, with the first element as a function, and the rest of the elements as extra arguments. So for example "`[f $x] $y`" is equivalent to "`f $x $y`". In this way we can use vectors as partially-applied functions, a common functional programming pattern.

4.2.4 Environments as Objects

Variable environments can be captured directly as key-value maps with a parent pointer. Environment capture uses `{ }` syntax, which runs a block of code in a new environment, then removes the running environment from its parent list. The resulting environment value is substituted in place of the `{ }` expression. Therefore all assignment syntaxes are available, including function definition. For example:

```
(make-account ?balance) = {
  balance = $balance
  (deposit ?amt) = (%balance = (add %amt %balance))
  (withdraw ?amt) = (%balance = (sub %amt %balance))
}

my-account = (make-account 10)
$my-account!deposit 20
put $my-account!balance # => 30
$my-account!withdraw 5
put $my-account!balance # => 25
```

A planned extension would allow using special syntax to register additional parents, allowing users to inherit by direct delegation:

```
(fancy-account ?b) = {
  +(account $b) # add the object to the lookup chain
  (display) = str "<account: " %balance ">"
}
```

4.2.5 Reading and Writing

Values are returned from functions by writing them to the standard output. They can be written by any function, but they can be directly written using the **put** builtin function. Asynchronous inputs (i.e. from a pipe) can be read using the **get** builtin function. Both of these block until they complete a communication.

4.2.6 Collection and Substitution

A priori, a capture mechanism such as described in Section 3.1.2 would have to return a list, as any process may output zero or more values. However, as a function calling convention, that would require callers to manually unwrap lists on every function call.

In order to simplify substitution, Magritte will collect and *expand* the values output by the contained code into the current command vector—increasing the argument number by the number of values output from the function. For example:

```
# A function definition: output three values
(count-three) = (put 1; put 2; put 3)

# Collect three writes and expand 1 2 3 in-place
other-fn 0 (count-three) 4
```

```
# equivalent to
other-fn 0 1 2 3 4
```

These semantics are similar to the `$(...)` syntax in bash, with the exception that we have the ability to properly separate values without relying on whitespace.

This mechanism is also available in vector literals, allowing us to collect outputs as a vector:

```
outputs = [(some-command)]
```

The `list` built-in function, which simply returns its argument vector, is also available for this purpose. The `for` function can also be used to splat vector arguments into function calls:

```
vec = [3 4]
some-fn 1 2 (for $vec)
```

4.2.7 Blocks

Blocks, on the other hand, do not collect or modify the environment's channels in any way, but instead simply run the code contained within them, and output values normally. These are mostly used to group commands within a pipeline, and in the body of function definitions:

```
generate-values | (process; process; process)
```

In the case that a user might want to use a substitution at the root level—i.e. to generate a function and immediately call it, we provide the `exec` builtin which executes its arguments as a command:

```
exec (put %put; put 1; put 2; put 3) # runs `put 1 2 3`
```

4.3 Concurrency and Interruption

4.3.1 Synchronous Channels

Channels in Magritte are **synchronous**—readers and writers cannot continue until a communication is completed successfully. Given the decision to allow rich values to be passed through channels, we have decided that a buffer is not as necessary for performance purposes, since a single write may contain an arbitrarily large amount of data—or for that matter a process handle or object reference. Additionally, synchronous channels have simpler semantics both for implementation and for users, and we leave open the possibility of user-implemented queues, such as:

```
producer | buffer 10 | consumer
```

4.3.2 Lazy Interruption

We have decided that the predictability of lazy interruption is worth the tradeoff for the extra-values problem discussed in Section 3.3.3. We plan to mitigate the extra-values problem in the future by providing a primitive `check` operation for a channel, which will interrupt the current process if the channel is closed, and be a no-op if it is open. In this way, function authors can opt-in to aborting early before performing expensive calculations that may not be used.

4.3.3 Process Registration

In order to satisfy the constraints of Section 3.3.4, we maintain a process register inside of each channel, so that we can decide when all readers or all writers have returned or been interrupted, at which time we can guarantee that processes on the other side of the channel cannot be woken up *without spawning new processes*. Therefore this covers the architecture of pipelines, in which many processes are spawned together in fixed configurations. Other architectures will have to manually manage process shutdown in some cases.

4.3.4 Compensation and Unconditional Compensation

We employ a variant of the compensation mechanism introduced by [*Inoue, Aotani, and Igarashi, 2018*] using the "`%%`" operator to indicate a compensation action, which is run in case of an interruption in the left hand side or subsequent lines. Compensations are cleared at the end of the current function body:

```
(my-function) = (
  action %% cleanup-action
  # in effect until the end of the function
)
```

Additionally, we define **unconditional compensations** using the "`%%!`" operator, which run both in the case of an interruption and also in the case of a normal return. In this way, they are analogous to `finally` or `ensure` sections of standard exception handling. For example, the function `read-lines` above could be implemented as:

```
(read-lines ?fname) = (
  f = (open-file $fname) %%! close-file $fname
  until (=> eof? $f) (=> read-until "\n" $f)
)
```

In this way, we can ensure that the file is closed when the function exits, whether by a normal return or by interruption.

4.3.5 Interrupt Handling

While we plan to explore more general mechanisms for exception handling, we find that it suffices for most applications to provide two builtin functions, `produce` and `consume`, to indicate the intent to fill or consume the entirety of the output or input streams, respectively. Each of these functions takes a single zero-argument function which will loop forever until the standard output or standard input respectively is closed, whereby control flow continues after the invocation. Using these functions, we might define `each` as:

```
(each ?fn) = (consume (=> %fn (get)))
```

With this definition, the call to `consume` will mask the interruption from the standard input closing, and control flow after any **`each`** invocation will continue as normal. This is enough to resolve the issue discussed in Section 3.3.7.

4.4 UNIX Integration: The Input-Stealing Problem

We determined that a fully-transparent design as described in Section 3.4 was not possible. However, we found a solution that was able to handle most cases, so long as the user follows the simple guideline that *external processes that use the standard input must be explicitly directed using a pipe or a redirect operator.*

The reason a fully transparent solution is more difficult than expected is a problem we call the **input-stealing problem**, in which use of external programs that do not use the standard input can nevertheless drain or "steal" inputs from the environment. To illustrate, consider the following code:

```
(f) = (
  shell echo "a debug statement!"
  each (?x => str "f processed: " $x)
)

put 1 2 3 | f
```

A user would expect this code to output four lines: a debug statement, and three lines with the string **"f processed: "** prepended. With a naïve implementation, however, the output will only contain the debug statement. Though the `echo` command does not actually open or read from its standard input, its input file descriptor has an empty and rather large **buffer**, and will report that it is currently writable. Thus the values from the standard input will be "stolen": serialized and written to the buffer, only to be discarded when `echo` exits.

We therefore will require a syntactic distinction between external commands that use the standard input and commands that don't. For external commands used in a pipeline, or external commands with an explicit redirect operator, this will work transparently. However, if a user wishes to implicitly use the standard input, they must make that use explicit:

```
# will not work: does not connect the input to `tr`
(capitalize) = tr a-z A-Z

# this will cause the input to be connected
(capitalize) = drain | tr a-z A-Z
```

This restriction allows us to avoid the input-stealing problem.

Chapter 5

Implementation of Magritte

In this chapter we will discuss various stages of the implementation of Magritte. The implementation is written in Ruby, and the source code is publicly available at `https://github.com/prg-titech/magritte`.

5.1 Magritte-DSL

We implemented Magritte in two steps: first, we implemented Magritte-DSL, a Ruby DSL that expressed the concurrency semantics of Magritte. Finding that the DSL was not alone sufficient, we then used Magritte-DSL to implement an interpreter for Magritte syntax. We will explain the embedded syntax, but use equivalent Magritte syntax in further examples for simplicity.

5.1.1 DSL Syntax

The Ruby DSL implemented the basic concurrency primitives of Magritte using Ruby blocks and `instance_eval`.

```
Magritte::DSL.run do
  s { put 1 }.p { put(5 + get) }.call
  put 10
end # => [6, 10]
```

The method `s`, globally available in DSL blocks, creates a "spawn" object, but does yet run any code. This is because we may decide to run it in a number of different ways later. In this example, we use the `.p { ... }` method to chain the resulting spawn object into a new one, such that the two pieces of code will be piped together. At the end, we call `.call`, which launches the threads and waits for the final one to finish. We could also have used the `.go` method, which is similar to the `&` operator in Magritte—it would run the code in the background, and continue without waiting for it to finish.

Alternatively, we could have used the `.collect` method, which would modify the environment of the contained code such that it didn't write to the outer output, but instead returned an array of its results:

```
Magritte::DSL.run do
  one, two, three = s { put 1; put 2; put 3 }.collect
  put two
end # => [2]
```

This system implements the process registration system described in Section 4.3.3, so it is also well-behaved on infinite loops:

```
Magritte::DSL.run do
  s { for_ [1, 2, 3] }
    .p { loop { put (get * 2) } }.call
  put 20
end # => [0, 2, 4, 20]
```

In this example, we use the "`for_`" method[1] to output all the elements of an `Enumerable` object, then pipe it into a component that runs an infinite loop, outputting twice whatever it reads from its input. When the `Enumerable` runs out of values, it will naturally return, causing channel cleanup. Then the call to `get` will result in a Ruby exception, which will unroll the stack, breaking the loop. Since the outer scope does not have the closed channel in its input, it then continues from "`put 20`".

5.1.2 Architecture

Magritte-DSL is broken into several modules:

- `Std`: A collection of shared methods, available in Magritte-DSL scopes. Contains methods such as `put`, `get`, `s`, and `make_channel`.
- `Spawn`: The class of the object returned by "`s { ... }`" in Magritte-DSL. It contains a Ruby block of code to run, along with an environment to run it in.
- `Proc`: A Magritte procedure, running concurrently with other procedures. Contains a reference to a Ruby thread running the code, and a stack of `Frame` objects. Responds to `.env`, to return the top frame's environment. Its class structure is shown in Figure 5.1. We use a thread-local variable to keep track of the currently-running `Proc`, accessible globally as `Proc.current`.
- `Frame`: A frame of execution. Contains an an `Environment`. These objects are registered to `Channel`, and de-register when the frame exists (see Section 5.1.2). A frame will also contain a list of `Compensations` to be run on interruption or return.
- `Channel`: A channel. Contains a two registries of reader and writer frames, respectively, and a set of blocked threads. Every thread in `@blocked_threads` is interrupted when the reader or writer registries become empty.
- `Env`: An environment. Contains an optional parent pointer, as well as lists of input and output channels (which default to the parent if missing). The interpreter extends this with a hash-map of variable names to `Ref` objects, which are passed around to enable mutation from other scopes. Includes a static `.base` method, creates a new `Env` including all entries from `Builtins` as well as everything defined in the special file `prelude.mag`.

Process Architecture

A process is represented by a `Proc` object, which contains a stack of `Frames`. Each time `.call` or `.collect` is used, we run the contained code not in its own process, but in a new frame on the current one. The frame contains an `Env` object, which keeps track of the current input and output channels. In the interpreter, it also keeps track of variable bindings.

[1] with an underscore because `for` is reserved in Ruby

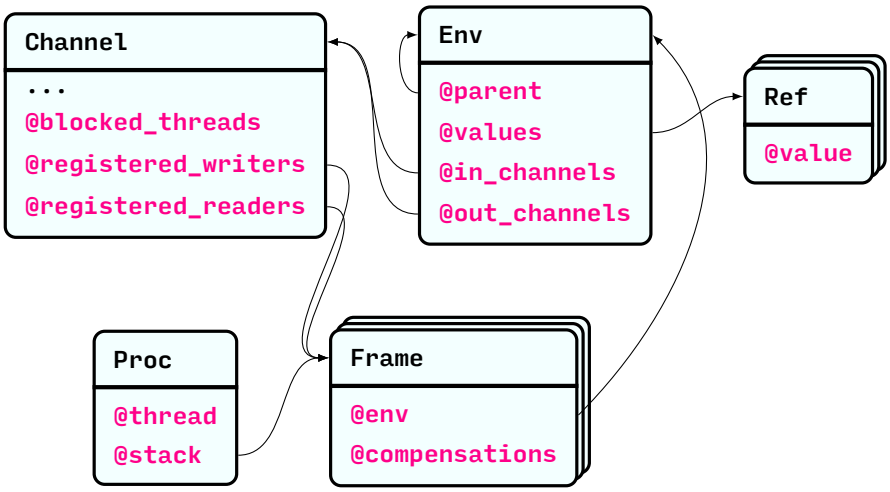

FIGURE 5.1: The Structure of a `Proc`

Channel Registry

Our channel implementation is a standard implementation of synchronous channels, with the addition of four intrinsic methods, used only internally by the interpreter: `add_reader`, `remove_reader`, `add_writer`, and `remove_writer`, which register and deregister processes as described in Section 4.3.3. When a `remove_*` method results in an empty set, it will additionally close the channel and raise an internal exception in every blocked thread, as shown in Figure 5.2.

Once the channel is closed, every call to `read` and `write` will interrupt the calling process as described in Section 3.3.5.

In order to reduce unnecessary use of Ruby threads, we also find it is simpler, instead of registering *processes* to the channels, to register *stack frames*. In this way, we can register all inputs and outputs on frame entry, and use standard Ruby exception handling to ensure we properly run compensations and deregister inputs and outputs on frame exit. This means that different frames in the same process can be connected to different channels, which makes the `>` and `<` redirection syntax straightforward to implement—we simply push a new frame with different channels attached.

Interruptions then cascade naturally—when a channel closes, a process is interrupted, causing it to unwind its stack and deregister channels, thereby potentially causing other channels to close.

Collectors

In order to implement the return semantics described in Section 3.1.2, we also implement a write-only channel called a *collector*, and an intrinsic that waits for channel closing. Collectors cannot be created directly by users, but only appear in the interpreter when we evaluate parentheses in argument position.

Naïvely, collectors would ignore registration commands and simply append written elements to an array. However, it is still necessary to track registered writers. Consider the example:

```
(range ?n) = (count-forever | take %n)
my-list = [(range 10 | (& drain; & drain))]
```

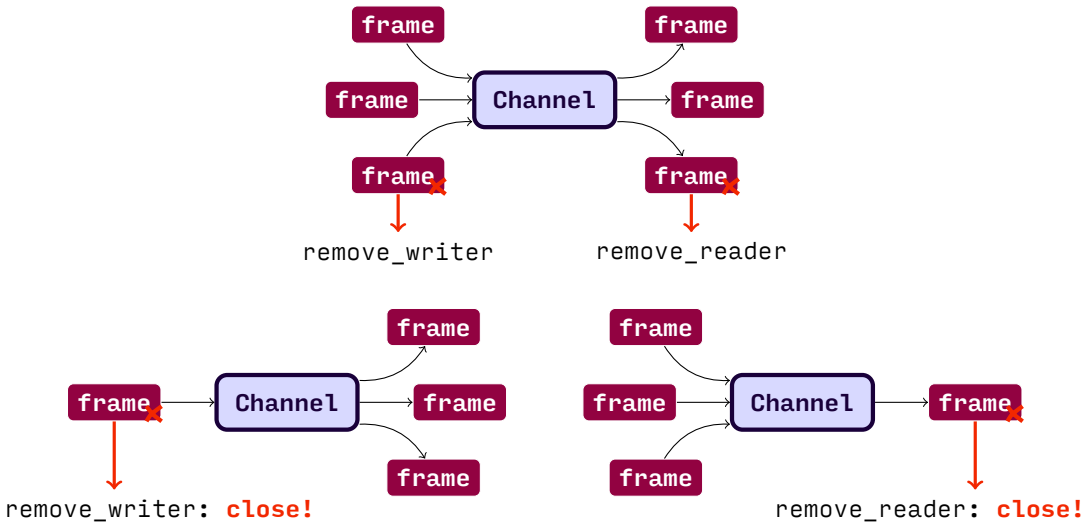

FIGURE 5.2: Channel Closing

In this example, there is an open question of how long we should wait until reading the collection and continuing. If we naïvely wait until the base command is finished, we will continue early and miss values that may be written later. Thus substitution waits until all writers to the collector have deregistered.

To implement this, we add a `wait_for_close` intrinsic to collectors, which returns immediately if closed, and otherwise adds the current thread to a waiting set and sleeps. Upon closing the channel, all elements of this waiting set are awoken, and flow continues. Thus the algorithm for substituting a block is:

```
* Create a new collector $c
* Run the parenthesized expressions with standard output set to the
    collector
* Run $c.wait_for_close
```

In the most common case, there will only be a single thread writing to the collector, so that the channel will be closed by the time the evaluation is finished, making `wait_for_close` a null operation.

Spawning Order Dependency

It is necessary to take some care with the implementation of the spawning primitive (`&`, or `.go`), that we wait until the spawned process has finished registering its channels before the spawning process continues. Consider the following example, which outputs 10 numbers, possibly out of order:

```
(drain) = (each (?x => put %x))
count-forever | (& drain; & drain) | take 10
```

The middle process is responsible for spawning two processes that funnel data from their input to their output. However, since they are both spawned in the background, the spawning process will immediately return. In general, the `drain` processes should be keeping the two pipes open. However, if we do not take care to wait until they have finished registering their channels, there is a risk that the spawning process will return first and close the two pipes.

Interrupt and Status

In order to implement the stack-unwinding behavior described in Section 4.3.5, we ensure that every interruption contains a `Status`, which indicates the intended effect of the interruption. Every use of `.call` also returns this status, and properties of the status are later used to implement `&&` and `||` in the interpreter. Each status also contains an optional reference to a `Reason` for the crash. This could include code errors such as type mismatches, or it could include a channel closure. The `Proc` runtime will inspect these reasons as they bubble up the stack, and stop the unwinding if interrupt was caused by a channel that has gone out of scope. Additionally, the `loop_channel` function from `Std` inspects these reasons and conditionally stops the unwinding. This is used to implement the `produce` and `consume` functions, which protect against `stdout` and `stdin` closure respectively.

5.1.3 Shortcomings

The main shortcoming of the embedded DSL was the fact that we could not modify Ruby's call, return, and variable binding semantics. To illustrate, consider this function which we used to implement a looping server (similar to the one shown in Section A.2):

```
def server_request(channel, message)
  receiver = make_channel
  s { put [message, receiver] }.into(channel).go
  s { put get }.from(receiver).collect[0]
end
```

In this example, we send a message and a reply-channel into a server channel and expect to read one message back. We will focus on the final line, in which the simple task of "read a value from this channel and return it", expressed in Magritte syntax as "**`get < $receiver`**", becomes much more complex due to the Ruby semantics surrounding it.

First, we must redirect the input to the receiver. To do this, we must create a new spawn context and use the `.from` method to specify the input channel:

```
s { get }.from(receiver).call
```

However, the behavior of the method `get` is to *return* the read value, which we have no way of accessing. So instead we must output it using `put`:

```
s { put get }.from(receiver).call
```

This will indeed output the correct value. However, it will write the output to the current `stdout`, rather than returning it. In Magritte, these are the same operation, but in Ruby, this means that the caller of `server_request` can no longer simply bind its result to a variable. Instead of `response = server_request(c, m)`, the caller would have to use a `.collect` operation to access the results:

```
response = s { server_request(c, m) }.collect[0]
```

Additionally, the `.collect` operation must return an array, which the caller must unwrap.

In Magritte, the parenthesis and bind semantics do this work for us, so the above would be equivalent to "`response = (server-request $c $m)`". However, in Ruby, we want to enable direct binding, so we have to patch this in the function body itself:

```
s { put get }.from(receiver).collect[0]
```

This example illustrates that the unification of return values with output is a necessary feature that is not in general implementable as a library. A language with its own semantics is necessary.

5.2 The Magritte Interpreter

5.2.1 Parsing

The syntax is described formally in Appendix B. We present a higher-level guide to the implementation here.

Skeleton Trees

We specify Magritte syntax using a parsing technique called **skeleton trees** [*Bachrach and Playford, 1999*], which allows enough flexiblity for both REPL use and extension with macros. The overview of the parsing system looks like:

$$\textit{Text} \rightarrow \textit{Tokens} \rightarrow \textit{Skeleton} \rightarrow \textit{AST}$$

Because of the extra step, we will describe more complex syntax forms by their translation from the skeleton tree data structure, which we introduce here.

The strategy begins with a normal lexer, which transforms the input text into a token stream. In order to ensure that all code is expressible in a one-dimensional manner (see Section 3.2.2), we completely unify the characters `;` and newline into a single `nl` token, so that both are transparently interchangeable throughout the language. These `nl` tokens are consolidated (so there are never two in a row), and filtered based on their previous and next tokens to allow line continuation.

The remaining `nl` tokens, along with nesting tokens such as "`(`" and "`)`", are used to break the tokens into groups and lines, represented by a **skeleton tree**.

This stripped-down syntax tree only has four node classes:

- `[root ...?nodes]` is a special node that specifies the root of the tree.
- `[tok ?type (?val)]` specifies an individual non-nesting token with an optional value. Most simple syntax elements, like variables, constants, and non-nesting punctuation are represented by this node.
- `[nested ?open ?close ...?elems]` specifies a nested group with open and close delimiters. All nesting tokens, such as `[ ]`, `( )`, and `{ }` have their contents grouped together in this type of node. In this way, the skeleton tree ensures that the input is well-nested.
- `[item ...?elems]` represents a newline-separated "item"—a semantic line, which is separated by `nl` tokens. This line-continuation behavior is controlled by defining **properties** on each token class (See Section B.2.2 for a detailed description).

For example, given the code

```
a = {
  (f ?x) = $x
  y =
    (add $z $w)
}
```

we would generate a tree similar[2] to:

```
[root
  [tok BARE "a"]
  [nested "{" "}"
    [item
      [nested "(" ")" [tok BARE f] [tok BIND x]]
      [tok EQL]
      [tok VAR x]]
    [item
      [tok BARE y]
      [nested "(" ")" [tok BARE add] [tok VAR z] [tok VAR w]]]]]
```

The AST is then generated recursively by pattern-matching on the skeleton tree. However, because the skeleton tree has already ensured and expressed the nesting of the code, this pattern-matching can be naïve and not worry about nested forms.

A macro in this system would be an operation *Skel* → *Skel*, such that the resulting tree is parseable as an AST. Importantly, this would only enforce that the *output* is valid syntax—the *input* only needs to be able to be broken into lines and groups, which would enable the free use of the Magritte token structure, independent of Magritte's actual syntax categories.

Lambda Clause Disambiguation

When parsing with traditional EBNF-like tools, it is often difficult to implement truly free lambda clauses. We managed to not require delimiters around multiline sub-clauses in Magritte through the use of skeleton trees.

In the first version of Magritte, we required that multi-statement lambda bodies be surrounded by parentheses, for fear that we would introduce ambiguity otherwise:

```
iter-tree = (
  [node ?l ?r] => (iter-tree $l
                   iter-tree $r)
  [leaf ?v] => put $v
)
```

However, as we developed a programming style for Magritte, we found that this created an unacceptable overhead for multiline, single-pattern functions. Simple expressions suddenly required two levels of parentheses: for example,

[2] Ideally, we would represent these using actual Magritte vectors, so as to allow manipulation with Magritte code. In the implementation, they are actually Ruby objects, and contain some extra information such as source location.

```
produce-values | each (?value => (
  f $value
  g $value
))
```

For this reason, we wanted to do away with the inner parentheses, and consider subsequent lines to continue the previous clause. However, since expressions and patterns often re-use the same tokens, it is difficult to tell where an expression stops and a new pattern begins. This grammar is ambiguous:

```
lambda ::= "(" clause* ")"
clause ::= pattern "=>" exp*
```

The ambiguity comes from the fact that the beginning of `pattern` and the end of `exp` share tokens. Most languages solve this by amending the syntax—either requiring a delimiter between clauses, or by requiring that the body be surrounded by delimiters.

Even when a delimiter between clauses is required, however, this can introduce problems if the same delimiter can function as a separator in the body. For example, in OCaml, this syntax is ambiguous:

```
let f = function
| A x -> match x with
         | C y -> ...
         | D z -> ...
| B -> ...
```

Because there is no delimiter at the end of a `match` expression, the OCaml parser is not able to tell whether "`| B -> ...`" represents a new clause in the inner `match` or the outer `function`. OCaml users are usually advised to surround the inner match with parentheses to avoid this ambiguity.

In Magritte, pattern matches are only performed with lambda functions, which already define their extent entirely using parentheses. We also use a separator token to separate clauses: the `nl` token, which can either be a real newline or a semicolon.

In our case, however, the skeleton tree parse already handles breaking the inside of the parentheses into lines. Thus our parser can simply search for lines that contain `=>` and combine the right-hand side with all the subsequent lines that do *not* contain `=>`. This combined set of lines is then parsed as a single group.

5.2.2 Interpretation

The interpreter builds on the primitives designed for Magritte-DSL by adding an interpreter and builtin-registry layer. It adds the following modules, as shown in Figure 5.3:

- `Interp`: The actual expression interpeter, implemented as a simple visitor over the AST. Includes `Std` to call into the concurrency APIs. Expresses all Magritte values as instances of a superclass `Value`. Though we cannot implement proper tail-call optimization due to the limitations of Ruby, we do detect tail calls (the last line in a group) and eagerly unregister the current `Frame`.
- `Builtins`: A registry of builtin functions—Magritte functions implemented in Ruby, using methods from `Std`.

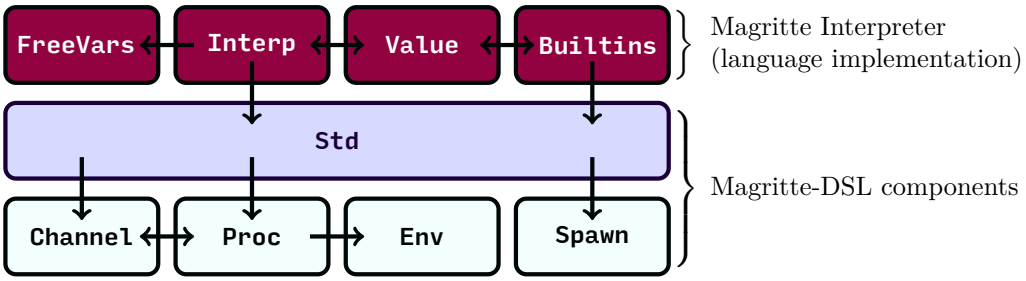

FIGURE 5.3: Implementation Architecture

- `Value`: A collection of classes to represent Magritte values. This includes the class `Function`, which calls back into the interpreter, and the class `BuiltinFunction`, which is a callable value that references one of the functions from `Builtins`.
- `FreeVars`: Statically scans an entire AST from the root, and creates a cache of free lexical variables. This is used at runtime by the interpreter to capture free lexical variables at function creation time. It also performs lifting of lambda declarations to enable mutual recursion, and replaces barewords in call position with lexical variable references.

5.3 Solutions to the Input Stealing Problem

We outline several considered solutions to the input-stealing problem.

5.3.1 Read detection

Our first attempt at solving the input-stealing problem involved detecting when the subprocess actually performs a read, and only then consuming an input from the Magritte standard input. Unfortunately, we were not able to find a UNIX API that was both correct and performant. Most methods, such as `select` and `fcntl`, will always indicate that a pipe is writable if the buffer is empty, regardless of whether any process is actively reading.

5.3.2 Write-Back

Another option would be to attempt to read back the contents of the buffer after the program has ended, and write them back to the Magritte standard-input. We implemented the following Ruby program as a proof of concept:

```ruby
# readback.rb
data = (1..10).map(&:to_s).join("\n")

stdin, inp = IO.pipe
outp, stdout = IO.pipe

pid = Process.spawn(*ARGV, STDIN => stdin, STDOUT => stdout)

Thread.new { inp << data; inp.close }

Process.wait(pid)
```

```
puts
puts "============"
puts "remaining data:"
puts stdin.read.inspect
```

This program will spawn any command and write the numbers 1 through 10, each followed by a newline, then read back whatever data was not consumed and print it to the console. For example:

```
$ ./readback.rb bash -c 'read; read; read'

============
remaining data:
"4\n5\n6\n7\n8\n9\n10"
```

This works by connecting the external process to a UNIX pipe (`IO.pipe`), and then *reading* from that pipe once the process has finished. We could theoretically keep track of how many bytes each Magritte value used, perform a calculation to determine which values from the Magritte standard input were unused, and write those values back to the channel.

Unfortunately, most UNIX programs assume that they have exclusive access to their standard input stream, and will greedily consume pages at a time of content, even for small reads. For example:

```
$ ./readback.rb bash -c 'head -3'

============
remaining data:
""

$ ./readback.rb ruby -e 'gets; gets; gets'

============
remaining data:
""
```

This means that functions that *do* read from the standard input will consume potentially many more values than was intended. We believe it is therefore necessary, for programs that use the standard input, to give the external program exclusive access to the input, and assume that the channel is completely consumed; and for programs that do not use the standard input, to never write any values at all. Unfortunately, as above, we have not found a generic way of making this distinction outside the kernel.

5.3.3 Special Functions “shell-in”, “shell-out”, and “shell-through”

Another option is to ask the user to specify whether their shell invocations will use the standard input, output, or both. For example:

```
shell-out echo hello        # uses stdout only
shell-in ./bin/consume-data # uses stdin only
shell-through tr a-zA-Z     # uses both stdout and stdin
```

This solves the input-stealing problem by specifying that when `shell-out` is used, no values are ever read from the Magritte standard input.

However, this places a large burden on the user, who will likely want to simply type `ls -l` and see results immediately, without worrying about its effect on the surrounding environment. Indeed, ideally we would not have such `shell` functions at all, but integrate a `$PATH`-based program lookup with the usual function lookup semantics in the interpreter.

5.3.4 Syntax Distinction

Our final design, which we have not yet implemented, works around the input-stealing problem by only allowing an external program to use the standard input when it is directly specified in the syntax, through the use of `|` or `<`. In this way, adding a simple debug statement will never use the standard input, but code such as:

```
sort %usernames | tr a-z A-z

./bin/consume-data < $channel
```

would allow the command to have exclusive access to the standard input. While this may be surprising behavior (as Magritte functions behave differently), we believe that this is less surprising than the input-stealing problem itself.

Chapter 6

Evaluation

In this chapter we will revisit the evaluation criteria defined in Section 1.5.

6.1 Magritte for Large Programs

6.1.1 Feature Sufficiency

We showed in Section 4.2 that the semantics of Magritte are sufficient to encompass many object-oriented programming patterns through the use of environments, as well as functional programming patterns through the use of vectors, lambda functions, pattern matching, and tail call optimization.

6.1.2 Example Programs

We further evaluated Magritte by implementing two simple programs.

Primality Test

We implemented the Miller-Rabin primality test [*Rabin, 1980*] using Magritte (code included in Section A.1). We selected this example because the standard implementation is described in a very imperative style, and we wanted to find out if the pipe-based style was expressive enough to represent this kind of algorithm without resorting to manual loops. Instead of loops, this code depends heavily on the standard library function `iter`, which is defined as:

```
(iter ?f ?v) = (produce (=> put %v; %v = %f %v))
```

This function is slightly dense, but in essence, it takes a function `?f` and an initial value `?v`, and runs an infinite loop in which it produces the value `%v` and then mutates it with the result of calling `%f %v`. In this way, instead of specifying a loop, we specify an infinite stream of values that we can map, filter, and limit.

For example, in the function `divmod-twos`, we wish to count how many factors of 2 are in a number; i.e., produce r and d such that $n = r \cdot 2^d$. To do this, we use `iter` to produce the sequence of pairs $(0, n), (1, \frac{n}{2}), (2, \frac{n}{4}), (3, \frac{n}{8}), \ldots$, then use `filter` and `get` to select the first entry in which the second element is odd. Finally we use `for` to unwrap the vector, and result in two outputs, which can be unpacked as r and d, respectively.

Later, in the definition of `test-prime`, we use the builtin function `produce`, which simply calls the given function in an infinite loop, to produce an infinite stream of random numbers, transform them, and limit them by `%i`. Instead of passing a function, however, we pass a vector, which when called will execute its first element as a command, as described in Section 4.2.3.

This code also makes use of the scope of parenthesized expressions to create a **module**: a collection of functions with a common closure. In this way, some helper functions are allowed to be private, and external code can access the two public members, `test-prime` and `gen-prime`, using object access syntax, for example `%miller-rabin!gen-prime`.

Stateful Server

This module (Section A.2) implements two main functions: `server`, and `send`. Through these three primitives, we implement a looping server similar to an actor (though its input is synchronous—it has no queue). In this way, we show that semantics similar to the actor model are possible, and that therefore process abstraction is possible.

The `server` function makes use of the `spawn` function, which will spawn a user-provided command vector. However, rather than inheriting the environment's input and output channels, it will create two new channels and immediately return them. In this way, we can start a program running in the background and interact with it freely through its input and output channels.

We must be careful, however, that we don't inadvertently close the channels before we are done. For example, a process started by `server` will receive messages from many different processes, which will immediately return, causing the number of registered writers to reach 0. Based on the automatic process cleanup semantics (see Section 3.3.4), this would close the server's input channel, terminating the server. To avoid this, we run a background process:

```
& sleep-forever > %i
```

Spawning this process ensures that the read channel of the server can only be closed by the server (the reading side), since there will always be at least one process that is a registered writer (even if it never actually performs a write). It will be naturally interrupted when the server shuts down.

The `send` method simply creates an anonymous reply-channel, writes a pair of the message and the reply-channel to the server, and drains the channel—waiting for the server to write and close the reply-channel. The `server` function ensures that the reply-channel will naturally become the standard output of the implementation function:

```
([?msg ?reply] => exec (for %fn) %msg > %reply)
```

We also include an implementation of the same program using Magritte-DSL. Here we find that not only is the syntax more cumbersome (requiring `s { ... }` around most expressions), but also that the problem discussed in Section 5.1.3 arises. That is, the `send` method must *collect* the reply-values into an array and return the array, rather than simply outputting them normally, as in the Magritte implementation.

We also find that the lack of pattern-matching makes the server implementation more cumbersome: it must manually destructure the message using ruby's `case` expression.

6.1.3 Discussion

While these two examples cannot be considered "large", they show that:

- Magritte is capable of basic modularization, namespacing, and private functions,

- the pipe-based programming style is capable of modeling loop-heavy imperative code, and
- Magritte is capable of using the pipe-based programming style to emulate some other concurrent abstractions.

Importantly, these are all difficult or impossible in bash and in most existing shell languages.

6.1.4 Drawbacks

Debugging

Debugging Magritte code proved to be difficult. Because the default composition method involves concurrent code, the resulting code is very concurrent and often a programmer will find themselves facing unexplainable deadlocks or race conditions for even simple code. This is partly due to the nature of concurrent programming, but partly also due to the lack of any kind of debugging support in the current system.

We believe that integration with a concurrent debugger such as Kómpos [*Marr et al., 2017*], would go a long way towards improving the usability of Magritte for more complex programs. Such a tool would allow operations like jump-to-read, where we could trace the flow of a single value through a channel to its reader, or break-on-channel-operation.

Speed

The current implementation is also very slow. This may be due to the speed of the Ruby runtime, but there is also overhead involved in using unbuffered channels for all composition. We believe a new virtual-machine based runtime, as well as support for multi-read and multi-write operations, could go a long ways towards improving the performance.

6.2 Magritte for Shell Scripting

Due to the input-stealing problem described in Section 4.4, further work is required for Magritte to be appropriate for shell scripting. However, we have preserved barewords (Section 3.2.3) and the basic I/O model and calling convention, so if we are able to solve the input-stealing problem and implement the marshalling system, Magritte will be very useful for scripts that combine unrelated programs.

6.3 Magritte as an Interactive System Shell

This use case also relies on the unimplemented UNIX integration. However, we have preserved the properties of linear composition (Section 3.2.1) and enabled one-dimensional code (Section 3.2.2), which make REPL use straightforward.

6.4 Discussion

While there are many unimplemented features still required for Magritte to be viable for its intended use case in real-world applications, we believe that Magritte represents a promising direction in shell-language design.

Chapter 7

Conclusion

We propose Magritte as a viable direction for improving shell languages. We will conclude by summarizing a comparison between Magritte's pipe system and the traditional UNIX shell system, followed by outlining possible future directions for this research.

7.1 Comparison To Bash

7.1.1 Private Variables, Modularization

We showed in Section 6.1.2 that Magritte is capable of basic modularization using closures. This is a feature that is prominently missing from `bash`, as we discussed in Section 2.2.3.

7.1.2 Functional Data Structures

We showed in Section 4.1.4 and Section 4.2.2 that Magritte is capable of programming with immutable variant data structures and pattern matching. We showed this was impossible in `bash` in Section 2.2.1.

7.1.3 Object-Oriented Programming

We showed in Section 4.2.4 that Magritte is capable of basic object-oriented patterns, and that a planned extension would enable most object-oriented design patterns. Again, we showed in Section 2.2.1 that this would be impossible in `bash`, because there is no way to access heap memory.

7.1.4 Pipe-Based Programming

In Section 6.1.2 we introduced **pipe-based programming**, a style of programming in which all iteration is managed by processing streams of data through pipes. We evaluated this programming style by implementing a fairly imperative algorithm using no recursion or manual loops, but only generating and consuming values.

We showed in Section 2.2.2 that this is technically possible in `bash`, up to a limit. As long as the data is well-behaved and easily separable, pipe-based programming is a viable way to design `bash` programs. However, we showed in Section 3.1.1 that as soon as the data becomes more complex—either concurrently managed or difficult to separate by a consistent delimiter—the pipe-based approach breaks down, and the programmer is forced to refactor their code. The consistency of Magritte's value pipes ensures that this does not occur in Magritte.

7.1.5 Interrupt Handling

In Section 3.3.2 and Section 4.3.4 we described a way to define context-sensitive error compensation to be able to enable the configuration of scoped cleanup actions. We showed in Section 2.2.4 that the exception method available to `bash` is not capable of these features.

7.2 Future Directions

While Magritte has a long way to go to be usable as a robust language, we believe that the overall design is capable of supporting large programs. The following represent what we believe are next steps toward the goal of being able to use Magritte in practical applications.

7.2.1 UNIX Integration

The UNIX integration strategy described in Section 4.4 and has not yet been implemented, and thus has not yet been evaluated from a practical sense. A next step would be to integrate this idea into the interpreter and test whether it meets the requirements of Section 3.4.

7.2.2 Debugging Support

Integration with a debugging system such as Kómpos [*Marr et al., 2017*] would go a long way toward making the user experience of Magritte less frustrating. Additionally, many of the subtle bugs and surprising behavior are a result of the instability and opacity of Ruby's thread API. Implementing a virtual-machine-based runtime with a custom scheduler would give much more insight into and control over the runtime.

7.2.3 Feature Completion

Magritte is still missing many features. For concurrency, it is still not possible to express multi-select, and without this feature it is difficult to implement programs such as a queue. There is generally a need for much more custom handling of channels. A planned extension to allow multiple registered inputs and outputs would be a start, but it is likely that it is necessary to generalize the current concurrency system to lower-level primitives.

Additionally, direct object delegation remains unimplemented. This feature would need to be fully designed, implemented, and evaluated on existing object-oriented design patterns.

7.2.4 Library & Module Support

Once the core language semantics are more or less feature-complete, there remains much work to be done to provide proper library, packaging, and module support.

Appendix A

Example Programs

A.1 Miller-Rabin Primality Test

```
miller-rabin = (
  # Factor a number n = (2^r)*d. Here we generate [r d] pairs,
  # incrementing r and halving d each time, detect when d becomes
  # odd, and then take and unwrap the first entry
  (divmod-twos ?n) =
    iter ([?r ?d] => put [(incr %r) (div 2 %d)]) [0 %n]
    | filter ([?r ?d] => is-odd %d)
    | for (get)

  # The Miller-Rabin primality test, using $i random numbers.
  # Succeeds if num is probably prime, fails if num is not prime.
  (test-prime ?i ?num) = (
    r d = (divmod-twos (decr %num))

    # Check if any of the first r elements in the sequence
    # x_{i+1} = x_i^2 mod n is equal to n-1
    (test-sequence ?x) = iter (?x => mod %num (pow 2 %x)) %x
                       | take %r
                       | any [%eq (dec %num)]

    # Check (r^d mod num) for i random numbers r between 2 and n-2.
    produce [rand-between 2 (sub 2 %num)]
      | each (?rand => mod %num (pow %d %rand))
      | take %i
      | any (?x => eq 1 %x || test-sequence %x)
  )

  (gen-odds ?start) = produce [rand-between $start (mul 2 $start)]
                    | each (?x => incr (mul 2 $x))

  (gen-primes ?b) = gen-odds $b
                  | each (?n => iter [%add 2] %n | take (div 2 $b))
                  | filter [%test-prime 10]

  # output the module
  put { test-prime = $test-prime
        gen-primes = $gen-primes }
)

(__main__) = %miller-rabin!gen-primes 7 | take 5
```

A.2 Stateful Server

```
# spawns a function in the background, returning its
# input and output channels
(spawn (?fn)) = (
  i = (make-channel)
  o = (make-channel)
  & exec (for %fn) < $i > $o
  put $i $o
)

# starts a request-reply server. usage:
# server (?msg => put reply)
(server (?fn)) = (
  i o = (spawn %each ([?msg ?reply] =>
    exec (for %fn) %msg > %reply
  ))

  # keep the channel alive on the read end
  & sleep-forever > $i

  put { request = %i }
)

# sends a message to a server, and waits for
# a reply. output: the reply.
(send ?server ?msg) = (
  c = (make-channel)
  & put [%msg $c] > %server!request
  drain < $c
)

(__main__) = (
  s = (server (
    [greet ?x] => str "hello " %x
    [die] => crash
  ))

  x = (send %s [greet world])
  put $x # => hello world
  send %s [greet jneen]
  send %s [die]
)
```

A.3 Stateful Server (Magritte-DSL)

```
module Server
  include Magritte::Std

  ServerInstance = Struct.new(:request)

  def spawn(&block)
    i = make_channel
    o = make_channel
    s(&block).into(o).from(i).go
    [i, o]
  end

  def server(&block)
    i, o = spawn do
      each do |(msg, reply)|
        s(&block).into(reply).call
      end
    end

    # keep the channel alive on the read end
    s { Thread.stop }.into(i).go

    ServerInstance.new(i)
  end

  def send(server, msg)
    c = make_channel
    s { put [msg, reply] }.into(server.request).go
    s { drain }.from(c).collect
  end

  def test
    s = server do |msg|
      case msg[0]
      when :greet then put "hello #{msg[1]}"
      when :die then crash!
      end
    end

    hello_world = send(s, [:greet, 'world'])[0]
    hello_jneen = send(s, [:greet, 'jneen'])[0]
    send(s, [:die])
  end
end
```

Appendix B

Syntax Description

B.1 Overview

We formally describe the syntax of Magritte at three levels: the lexical structure (tokens), the skeleton syntax tree (skeleton nodes), and the final abstract syntax tree (AST). We also provide a small grammar detailing the *Tokens* → *Skel* phase of the parse, and a set of pattern-matches which implements the *Skel* → *AST* phase.

B.2 Lexical Syntax

B.2.1 Table of tokens

Name	Example	Description	Properties
Special Tokens			
`eof`	`<eof>`	end of file	`eof`
`nl`	`;`, `<\n>`	newline & semicolon	`nl`, `skip`
Nesting Tokens			
`lparen`	`(`	left parenthesis	`nested(rparen)`, `skip`
`rparen`	`)`	right parenthesis	`continue`
`lbrack`	`[`	left bracket	`nested(rbrack)`, `free_nl`
`rparen`	`]`	right bracket	–
`lbrace`	`{`	left brace	`nested(rbrace)`, `skip`
`rbrace`	`}`	right brace	`continue`
Punctuation			
`arrow`	`=>`	arrow	`skip`, `continue`
`equal`	`=`	equal sign	`skip`, `continue`
`write_to`	`>`	output redirect	`skip`, `continue`
`read_from`	`<`	input redirect	`skip`, `continue`
`d_per`	`%%!`	compensation	`skip`, `continue`
`d_per_bang`	`%%!`	uncond. compensation	`skip`, `continue`
`d_amp`	`&&`	conditional `and`	`skip`, `continue`
`d_bar`	`||`	conditional `or`	`skip`, `continue`
`d_amp`	`&`	spawn operator	–
`d_pipe`	`|`	pipe operator	`skip`, `continue`
`d_bang`	`!`	lookup operator	–
Variables & Constants			
`bind`	`?x`	variable binders	–

`variable`	`$x`	dynamic variables	–
`lex_variable`	`%x`	lexical variables	–
`intrinsic`	`@!x`	intrinsics (not in use)	–
`keyword`	`@x`	keywords (not in use)	–
`num`	`123`, `123.4`	numbers	–
`string`	`"foo"`, `'foo'`	strings	–
`bare`	`x`	bare words	–

B.2.2 Description of Token Properties

Name	Description
`eof`	Indicates the end of the file
`nl`	Indicates a newline token (may not be an actual newline). Should not appear in the skeleton tree.
`nested(?token)`	Indicates a nested pair to the skeleton tree parser. Should only appear as delimiters in `[nested ...]` elements in the skeleton tree.
`skip`	Erase any subsequent `nl` tokens
`continue`	Erase any *previous* `nl` token (implemented by peeking the next token in the skeleton tree parser)
`free_nl`	For a nesting token, ignore any `nl` tokens at the top level

The token stream is pre-processed to filter out `nl` tokens that appear after a token with the `skip` property, or before a token with the `continue` property.

B.3 Skeleton Syntax

B.3.1 Table of Nodes

Structure	Description
`[root ...?nodes]`	The root of the tree
`[nested ?open ?close ...?nodes]`	A nested structure surrounded by tokens with the `nested` property
`[item ...?nodes]`	A newline-separated "semantic line"
`[tok ?token]`	Any other token

B.3.2 Skeleton Syntax Grammar

The grammar below is included for reference. The implementation is manual recursive descent. Here we manually specify each nesting token, but in the implementation this behavior is defined by the `nested` and `free_nl` properties of the nesting tokens. The resulting tree will not to contain any `TOK` nodes that refer to `nl` or nesting tokens.

```
ITEM ::= (NESTED | TOK)+
NESTED ::= "(" ITEM*:nl ")" | "{" ITEM*:nl "}"
         | "[" (NESTED | ATOM)+ "]"
TOK ::= (* all non-nested, non-special tokens *)
ROOT ::= ITEM*
```

B.4 Abstract Syntax

B.4.1 Description of AST

We describe each node as `[NodeName ...?attributes]`. Recursive attributes (that are themselves nodes) are parenthesized. Attributes that are lists of nodes are presented as `[?attr]`. Optional attributes are presented as `{?attr}`. All other attributes are plain non-recursive data.

Structure	Example	Description
`[Variable ?name]`	`$x`	Variables
`[LexVariable ?name]`	`%x`	Lexical variables
`[Binder ?name]`	`?x`	Variable binders
`[String ?value]`	`"foo"`	Strings
`[Number ?value]`	`123`	Numbers
`[Vector [?els]]`	`[a b c]`	A vector.
`[StringPattern ?value]`	`foo`	A pattern that matches a string

`[VectorPattern`
`  [?patterns]`
`  {?rest}]`
	`[tag ?value]`
	~ or ~
	`[?first (?rest)]`
		A pattern that matches a vector, containing sub-patterns. `?rest`, if present, must be a `RestPattern`.

`[DefaultPattern]`	`_`	A pattern that matches nothing.
`[RestPattern ?name]`	`(?rest)`	A pattern that matches and collects the remaining elements of a vector

`[Lambda`
`  ?name`
`  [?patterns]`
`  [?bodies]`
`  ?range]`
	`(?x = 1; ?y => 2)`
	~ or ~
	`(?x ?y =>`
	`  ...`
	`  ...)`
		A lambda expression that maps a group of patterns to bodies. `?patterns` and `?bodies` must be the same length, and corresponding indices are matched. New clauses start at the beginning of lines that contain a `=>` token. We record an optional name and source location for traceback purposes.

`[Pipe (?producer) (?consumer)]`	`produce-values | consume-values`	A pipe expression. Nested left-recursively, so that `a | b | c` is similar to `(a | b) | c`.
`[Redirect ?direction (?target)]`	`> $channel` *~ or ~* `< $channel`	A redirection of input or output. Target must be a variable or lexical variable.
`[With [?redirects] (?expr)]`	`cmd > $out < $in`	Redirects inputs and outputs. `?redirects` must be a list of `Redirect` nodes.
`[Or (?lhs) (?rhs)]`	`c1 || c2`	Short-circuiting **or** operator
`[And (?lhs) (?rhs)]`	`c1 && c2`	Short-circuiting **and** operator
`[Else (?cond) (?rhs)]`	`c1 && c2 !! c3` *~ or ~* `c1 || c2 !! c3`	Specifies an **else** to continue a conditional. `?cond` must be an `And` or `Or` node.
`[Compensation (?expr) (?compensation) ?range ?is-unconditional]`	`c1 %% c2` *~ or ~* `c1 %%! c2`	Registers a compensation (See Section 4.3.4). A boolean specifies whether it is conditional. We also record a source location for traceback purposes.
`[Spawn (?expr)]`	`& fn`	Spawn operator: evaluates the expression in a new thread.
`[Command [?vec] ?range]`	`cmd arg arg arg`	A command vector. `?vec` may not be empty. A source location is recorded for traceback purposes.
`[Group [?els]]`	`cmd; cmd; cmd`	A generic group of commands.
`[Block (?group)]`	`(cmd; cmd; cmd)`	A group of commands to be run in a sub-environment.
`[Subst (?group)]`	`foo (cmd; cmd)`	A group of commands to substitute with their output
`[Environment (?group)]`	`{a = 1; b = 2}`	A group of commands and assignments to run, and substitute with the resulting environment.

`[Access (?source) (?lookup)]`	`$foo!bar` ~ *or* ~ `$foo!$bar`	Environment lookup. Can be used as an LValue.
`[Assignment (?lval) (?rval)]`	`x = 1` ~ *or* ~ `$x = 2` ~ *or* ~ `$o!x = 3`	An assignment or a mutation. The semantics depend on the syntax class of `?lval`.

B.4.2 Description of Parsing Rules

We describe the parsing rules ($Skel \rightarrow AST$) recursively, using a matching system similar to Macro By Example [*Kohlbecker and Wand, 1987*]. Any variable with a * on the left side should be matched on the right side, so that for example $F(\texttt{[}e^*\texttt{]}) = \texttt{[}G(e)^*\texttt{]}$ describes a mapping operation, in which every matched e on the left is replaced with $G(e)$ on the right. We define the * operator as matching **greedily**: that is, it will always match the largest possible number of elements. However, it is occasionally necessary to indicate whether we search from the left or from the right, and in this case we will use the symbol $^{*?}$ to indicate a **non-greedy** match. Thus $a^{*?}\ b\ c^*$ will find the *first* pattern that matches b, and $a^*\ b\ c^*$ will match the *last*[1]. The rules are also listed here in order—in the case that more than one rule applies, we always select the first one.

Occasionally, for simplicity, we simply write the text representation of a token instead of the `tok` skeleton node: for example, we will write `&&` instead of `[tok d_and]`.

Root

Root:
$MAG(\texttt{[root}\ e^*\texttt{]}) = GRP(e^*)$
Group:
$GRP(c^*) = \texttt{[Group}\ LIN(c)^*\texttt{]}$

Assignments & Commands

Spawn: `& f $x`
$LIN(\texttt{[item \&}\ v^*\texttt{]}) = \texttt{[Spawn}\ LIN(v^*)\texttt{]}$
Function Definitions: `(f ?x) = y z`
$LIN\big(\texttt{[item [nested "(" ")"}\ a\ p^*\texttt{] =}\ c^*\texttt{]}\big)$
Assignment: `x y = z w`
$LIN(\texttt{[item}\ l^{*?}\ \texttt{[tok =]}\ r^*\texttt{]}) = \texttt{[Assign [}LV(l)^*\texttt{] [}EXP(r)^*\texttt{]]}$
$= \texttt{[Assign [}LV(a)\texttt{] [Lambda [}PAT(p^*)\texttt{] [}EXP(\texttt{[item}\ c^*\texttt{])]]]}$
And/Or: `a && b, a || b`
$LIN(\texttt{[item}\ c^{*?}\ (o : \texttt{\&\&}|\texttt{||})\ d^*\texttt{]})$
$= ELS(T(o)\ \texttt{[item}\ c^*\texttt{]}\ \texttt{[item}\ d^*\texttt{]})$
where $T(\texttt{\&\&}) = \texttt{And}$ and $T(\texttt{||}) = \texttt{Or}$
Else: `a && b !! c`

[1] In the actual implementation, we have special matching primitives `lsplit` and `rsplit` for this purpose.

$ELS(o\ c\ \texttt{[item}\ (t|t \notin \{\texttt{\&\& ||}\})^{*?}\ \texttt{!!}\ e^*\texttt{]})$
$= \texttt{[Else [}o\ LIN(c)\ LIN(\texttt{[item}\ t^*\texttt{]})\texttt{]}\ LIN(\texttt{[item}\ e^*\texttt{]})\texttt{]}$

No-Else: `a && b, a || b`
$ELS(o\ c\ t) = \texttt{[}o\ LIN(c)\ LIN(t)\texttt{]}$

Compensation: `c1 %% c2`
$LIN(\texttt{[item}\ e^{*?}\ \texttt{\%\%}\ c^*\texttt{]})$
$= \texttt{[Compensation}\ LIN(\texttt{[item}\ e^*\texttt{]})\ LIN(\texttt{[item}\ c^*\texttt{]})\ \texttt{false]}$

Unconditional Compensation: `c1 %%! c2`
$LIN(\texttt{[item}\ e^{*?}\ \texttt{\%\%!}\ c^*\texttt{]})$
$= \texttt{[Compensation}\ LIN(\texttt{[item}\ e^*\texttt{]})\ LIN(\texttt{[item}\ c^*\texttt{]})\ \texttt{true]}$

Pipe: `c1 | c2 | c3`
$LIN(\texttt{[item}\ l^*\ \texttt{|}\ r^{*?}\texttt{]}) = \texttt{[Pipe}\ LIN(\texttt{[item}\ l^*\texttt{]})\ CMD(r^*)\texttt{]}$

Command: `f $x`
$LIN(\texttt{[item}\ e^*\texttt{]}) = CMD(e^*)$

Commands & Elements

With: `f $x < $c1 > $c2`
$CMD(e^{*?}\ (d : \texttt{<}|\texttt{>})\ c)^*)$
$= \texttt{[With [([Redirect}\ d\ EXP(c)\texttt{])]}^*\ CMD(e^*)\texttt{]}$

Elements
$CMD(e^*) = \texttt{[Command [}ELE(e^*)\texttt{]]}$

Access: `$foo!bar`
$ELE(e\ \texttt{!}\ n\ r^*) = \texttt{[Access}\ EXP(e)\ EXP(n)\texttt{]}\ ELE(r^*)$

Expr
$ELE(e\ r^*) = EXP(e)\ ELE(r^*)$

Empty Elements
$ELE() = \varnothing$

Variables & Constants

Variables: `$x`
$EXP(\texttt{[tok var(}n\texttt{)]}) = \texttt{[Variable}\ n\texttt{]}$

Lexical Variables: `%x`
$EXP(\texttt{[tok lex\_variable(}n\texttt{)]}) = \texttt{[LexVariable}\ n\texttt{]}$

Barewords: `foo`
$EXP(\texttt{[tok bare(}v\texttt{)]}) = \texttt{[String}\ n\texttt{]}$

Strings: `"foo"`
$EXP(\texttt{[tok string(}v\texttt{)]}) = \texttt{[String}\ v\texttt{]}$

Numbers: `123`
$EXP(\texttt{[tok num(}v\texttt{)]}) = \texttt{[Number}\ v\texttt{]}$

Nested Expressions

Vectors: `[a b c]`
$EXP(\texttt{[nested "[" "]"}\ e^*\texttt{]}) = \texttt{[Vector [}EXP(e)^*\texttt{]]}$

Lambda: `(x => y; z; w => u)`
$EXP(\texttt{[nested "(" ")" ([item}\ p^{*?}\ \texttt{=>}\ h^*\texttt{]}\ c^{*?})^*\texttt{]})$
$= \texttt{[Lambda [}PAT(p^*)^*\texttt{]}\ \texttt{[}GRP(\texttt{[item}\ h^*\texttt{]}\ c^*)^*\texttt{]]}$

Example 1:
$EXP(SKEL(\texttt{(x => y; z; w => u)}))$
$= EXP(\texttt{[nested "(" ")" [item x => y] [item z] [item w => u]]})$

$= \texttt{[Lambda [}PAT(\texttt{x})\ PAT(\texttt{w})\texttt{] [}GRP(\texttt{y; z})\ GRP(\texttt{u})\texttt{]]}$

Example 2:

$EXP(SKEL(\texttt{(x => y; z; w)}))$

$= EXP(\texttt{[nested "(" ")" [item x => y] [item z] [item w]]})$

$= \texttt{[Lambda [}PAT(\texttt{x})\texttt{] [}GRP(\texttt{y; z; w})\texttt{]]}$

Substitution: `(a b c)`

$EXP(\texttt{[nested "(" ")" } e^*\texttt{]}) = \texttt{[Subst }GRP(e^*)\texttt{]}$

Environment: `{a = 1; b = 2}`

$EXP(\texttt{[nested "\{" "\}" } e^*\texttt{]}) = \texttt{[Environment }GRP(e)^*\texttt{]}$

Patterns

Rest Pattern: `?x ?y (?z)`

$PAT(e^*\ \texttt{[nested "(" ")" [tok bind(}n\texttt{)]]}) =$

$= \texttt{[VectorPattern [}PTM(e)^*\texttt{] [Binder }r\texttt{]]}$

Pattern: `?x ?y`

$PAT(e^*) = \texttt{[VectorPattern [}PTM(e)^*\texttt{]]}$

Binders: `?x`

$PTM(\texttt{[tok bind(}name\texttt{)]}) = \texttt{[Binder }name\texttt{]}$

Vector Pattern Terms (`[?x ?y (?rest)]`)

$PTM(\texttt{[nested "[" "]" } e^*\texttt{]}) = PAT(e^*)$

Default pattern: `_`

$PTM(\texttt{[tok bare(\_)]}) = \texttt{[DefaultPattern]}$

Bibliography

Bachrach, J. and K. Playford (1999). *D-expressions: Lisp power, Dylan style*. Retrieved Dec 2018 from https://people.csail.mit.edu/jrb/Projects/dexprs.pdf.

Blandy, Jim (1998). "Guile: An Interpreter Core for Complete Applications". In: *Handboook of Programming Languages, 1st ed.* Ed. by Peter H. Salus. Vol. IV: Functional and Logic Programming Languages. Macmillan Technical Publishing, pp. 87–104. URL: http://gnu.org/s/guile.

Chu, Andy (July 2019). *Oil: A New Unix Shell*. Retrieved Jul 2019 from http://oilshell.org/.

Duff, Tom (1990). "RC—a Shell for Plan 9 and UNIX". In: *UNIX Vol. II*. Ed. by A. G. Hume and M. D. McIlroy. Philadelphia, PA, USA: W. B. Saunders Company, pp. 283–296. ISBN: 0-03-047529-5.

Graham, Paul (May 2002). *Revenge of the Nerds*. Retrieved Jul 2019. URL: http://www.paulgraham.com/icad.html.

Haahr, Paul and Byron Rakitzis (1993). "Es: A shell with higher-order functions". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, pp. 53–62.

Inoue, Hiroaki, Tomoyuki Aotani, and Atsushi Igarashi (2018). "ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Ed. by Todd Millstein. Vol. 109. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2:1–2:33. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.2. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9207.

Kohlbecker, E. E. and M. Wand (1987). "Macro-by-example: Deriving Syntactic Transformations from Their Specifications". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87. Munich, West Germany: ACM, pp. 77–84. ISBN: 0-89791-215-2. DOI: 10.1145/41625.41632. URL: http://doi.acm.org/10.1145/41625.41632.

Linux man-pages Project (Apr. 2018). *pipe(7) Linux User's Manual*. 4.16. The Linux Foundation. https://www.kernel.org/doc/man-pages/.

Mahoney, Michael S et al. (1989). *The UNIX Oral History Project: Release 0, The Beginning*. Retrieved Jul 2019. URL: http://www.princeton.edu/~hos/Mahoney/expotape.htm.

Marr, Stefan et al. (Oct. 2017). "A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS'17. (acceptance rate 64%). Vancouver, Canada: ACM, pp. 3–14. ISBN: 978-1-4503-5526-1/17/10. DOI: 10.1145/3133841.3133842.

Ousterhout, John K. (1989). *Tcl: An Embeddable Command Language*. Tech. rep. Berkeley, CA, USA.

Rabin, Michael O (1980). "Probabilistic algorithm for testing primality". In: *Journal of Number Theory* 12.1, pp. 128 –138. ISSN: 0022-314X. DOI: https://doi.org/10.1016/0022-314X(80)90084-0. URL: http://www.sciencedirect.com/science/article/pii/0022314X80900840.

Scopatz, Anthony (Dec. 2018). *The Xonsh Shell*. Retrieved Dec 2018 from https://xon.sh/.

Shivers, Orin (Jan. 1994). *A Scheme Shell*. Technical Report MIT/LCS/TR-635.

Xiao, Qi et al. (July 2019). *Elvish Shell: a friendly interactive shell and an expressive programming language*. Retrieved Jul 2019 from http://elv.sh/ and http://github.com/elves/elvish.