CHOC:
A COMMAND AND
PROGRAMMING LANGUAGE

by
Michael MacInnis

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

MASTER OF COMPUTER SCIENCE

Ottawa-Carleton Institute for Computer Science
School of Computer Science

CARLETON UNIVERSITY
Ottawa, Ontario

© 2010, Michael MacInnis

Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-68633-1
Our file *Notre référence*
ISBN: 978-0-494-68633-1

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

■ ■ ■
Canada

Abstract

This thesis presents Choc, a concurrent, higher-order, object-based command language. Choc forms the basis for the Choc shell, a reimagining of the Unix shell. The Choc shell retains the familiar look and feel of current Unix shells while making considerable improvements to the shell as a programming language.

Acknowledgements

Thanks to my supervisor, Dr. Douglas Howe, for his comments, suggestions and constructive criticism throughout the evolution of the Choc shell's design.

Thanks to my family. Without the love and support of my wife, Angelika, and daughters, Ava and Isabella, none of this would have been possible.

Table of Contents

Abstract	ii
Acknowledgements	iii
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Contributions	2
1.4 Organization	4
Chapter 2 Background	6
2.1 The Unix Shell	6
2.2 S-Expressions	9
2.3 The Π -Calculus	9
2.4 Prototype-Based Programming	11
Chapter 3 Related Work	13
3.1 Previous Attempts to Build a Better Shell	13
3.2 Scripting Languages	15
3.3 Lisp and the Unix Shell	17
3.4 Pict: A Programming Language based on the Π -Calculus	18
3.5 SELF and JavaScript	20
Chapter 4 Choc	21
4.1 Syntax	21
4.1.1 T-Expressions	22
4.1.2 Interactive and Historical Constraints	25
4.2 Semantics	28

4.2.1	Improved Programming Language Features	28
4.2.2	Improved Support for Concurrency	37
4.2.3	Improved Support for Abstraction	44
4.2.4	Improved Support for Modularity	45
4.3	Pragmatics	47
4.3.1	Command-Line Structure	48
4.3.2	Metacharacters	51
4.3.3	Creating New Commands	52
4.3.4	Command Arguments and Parameters	54
4.3.5	Program Output as Arguments	55
4.3.6	Objects	55
Chapter 5	Implementation Details	59
5.1	Storage Management	59
5.2	Processes and Channels	60
5.3	The Choc Interpreter	61
5.4	Current Limitations	61
Chapter 6	Conclusion	62
6.1	Results	62
6.2	Future Work	65
Appendix A	Language Specification	67
A.1	Words	67
A.2	Grammar	67
A.3	Commands	70
Appendix B	Source Code	76
B.1	License	76
B.2	Common	77
B.3	Storage	81

B.4 Parser	101
B.5 Scanner	109
B.6 Engine	115
B.7 Interface	169
Bibliography	172

Chapter 1

Introduction

The Unix shell, in one or more of its various incarnations, remains a staple in the toolbox of many programmers. Combining a command and programming language is a powerful concept [13]. The Unix shell in particular offers constructs found in few other languages and although the various incarnations differ in detail, the fundamental character of the Unix shell has remained largely unchanged for more than 30 years.

Unfortunately, part of the reason for the Unix shell's unchanging character is neglect. Described by those closest to it as a strange language, the Unix shell is often criticized for its ad hoc design and weak programming facilities [20, 39]. Worse, the Unix shell has developed a reputation for being resistant to innovation in programming languages and this view is largely accepted as an unavoidable consequence of its dual constraints as both a programming and command language [15, 20].

1.1 Motivation

Command languages existed before the Unix shell but few with the same power and flexibility. The Unix shell also introduced and popularized the concept of pipes [42] - an elegant and interesting construct rarely found in other languages. Unix shells have been called a zenith of programming language design but, more often, they are called a nadir of programming language design [23].

The promise of the shell - a combined command and programming language - has gone mostly unfulfilled. As we will demonstrate, Unix shells are overly simplistic languages. Their programming language features are ad hoc and in many cases artificially restricted. Most attempts to improve the shell as a programming

language have failed and so the programming language features of Unix shells have stagnated. Unix shells have developed a reputation as horrible programming languages [39].

1.2 Problem Statement

Are the Unix shell's constraints as a command language so severe that the programming language features must inevitably suffer? We claim that, although the Unix shell's interactive constraints are numerous, it is possible to satisfy these constraints while improving the programming language features of the Unix shell. The Choc shell was developed as proof of that claim.

1.3 Contributions

The Choc shell is a freely available shell released under an MIT-style license that retains the look and feel of current Unix shells while adding first-class channels, methods and environments and support for concurrent, higher-order and object-based programming. Choc was implemented as proof that a combined command and programming language can satisfy the Unix shell's interactive and historical constraints while incorporating advanced programming language features.

The Choc shell was inspired by previous attempts to combine Lisp and the Unix shell [12, 39]. Unlike previous attempts, however, Choc is not implemented by embedding a Unix shell in Lisp. Instead Choc was designed from scratch so that it could not only incorporate features from Lisp but also modify and correct those characteristics that do not interact well with the constraints imposed on the Choc shell as a command language.

The Choc shell was also motivated by the belief that many of the flaws in current Unix shells are not inherent but rather historical. Design choices that are clearly unfortunate in retrospect have been carried forward by current Unix shells in the name of backward compatibility. As a reimagining of the Unix shell, the Choc shell revisits these design choices.

Pipes, which are central to the Unix shell, are widely recognized as a simple and elegant way to compose programs, but Unix shells have been criticized for restricting this form of communication to unstructured byte streams. The Choc shell embraces and extends this aspect of the shell by introducing t-expressions, a unified format for the representation of code and data. This allows data to be marshalled and unmarshalled easily, shared between processes and piped from, to or even through external programs.

The Unix shell is a concurrent language. It would seem natural to base a Unix shell on a concurrent model of computation, like the π -calculus [27]. The emphasis on processes communicating over channels, in the π -calculus, corresponds nicely with the Unix shell's focus on processes connected by pipes, but the dynamics of the π -calculus describe concurrent operations from a very different point of view. To solve this mismatch we introduce a variant of the π -calculus which includes sequential composition as a primitive operation and use this variant to provide a solid theoretical foundation for the constructs provided by the Choc shell.

As a Unix shell, the Choc shell is both a command and programming language but, for Choc to be a serious programming language it needs mechanisms to support modularity. Choc introduces a novel form of environment and exposes these environments as first-class values. By exposing environments as first-class values Choc can unify the treatment of what is normally a number of limited and specialized mechanisms and support a powerful and flexible form of object-oriented programming.

To summarize, the Choc shell makes the following contributions:

- The Choc shell improves the programming language features of the Unix shell and in doing so demonstrates that many of the Unix shell's problems as a programming language are not inherent but rather historical.
- The Choc shell demonstrates that Lisp and the Unix shell can be successfully combined. The Choc shell incorporates features from the Scheme dialect of Lisp while maintaining the look and feel of current Unix shells.

- Choc introduces t-expressions - a syntax for representing semi-structured data, which Choc uses as a unified format for both data and programs. T-expressions enable data to be marshalled and unmarshalled easily, shared between processes and piped from, to or even through external programs. T-expressions are fundamental in allowing channels to be implemented as pipes, which allows for better integration with Unix and existing Unix tools. T-expressions also enable more uniform semantics when passing arguments to internal methods and external programs.
- Choc introduces a variant of the π -calculus which includes sequential composition as a primitive operation. Choc uses this as its underlying model of computation, providing a solid theoretical foundation for Choc and the Choc shell.
- Choc introduces a novel form of first-class environment. These enable Choc to unify the treatment of objects and modules and support a flexible form of object-oriented programming that complements the shell's interactive nature.

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 reviews the history of the Unix shell, and briefly describes s-expressions, the π -calculus, and prototype-based programming - all of which are foundational to the design of Choc. Chapter 3 discusses other attempts to build a better Unix shell including attempts to add shell-like features to existing scripting languages, Lisp and the Scheme dialect of Lisp. Also covered in chapter 3 are the languages Pict[33], SELF[2] and JavaScript[50]. Choc is related to these languages by virtue of its being based on the π -calculus (Pict) and because of its prototype-based object system (SELF and JavaScript). Chapter 4 introduces Choc. Chapter 5 discusses Choc's current implementation. Results and future work are presented in Chapter 6. Choc's language specification and the source code for Choc are included as

appendices.

Chapter 2

Background

2.1 The Unix Shell

Often referred to in the singular, the term Unix shell actually refers to a family of shells. The earliest Unix shell was the Thompson shell. Created by Ken Thompson, it was released in the early 70s, with versions 1 through 6 of the Unix operating system. While many key concepts were already present, the programming features were rudimentary. The only flow control constructs present in the Thompson shell were `goto` and `if` and these were implemented as separate commands, external to the shell [28].

In 1975 the PWB (Programmer's Workbench) shell, also known as the Mashey shell, after its creator John Mashey, was released. It made improvements to the Thompson shell to allow for more serious shell programming. In particular, the PWB shell added rudimentary variables and included the `if`, `switch`, `while`, and `goto` control structures as shell built-ins [24].

In 1976 Steve Bourne began working on a redesign of the Unix shell. The Bourne shell was implemented from scratch. It was incompatible with the earlier Thompson and PWB shells but included equivalents for the features found in these earlier shells. The Bourne shell introduced environment variables and elevated scripts to the same status as other commands [48, 51]. In 1979 with the release of version 7 of the UNIX operating system, the Bourne shell (`sh`) became the standard Unix shell.

Meanwhile, at UC Berkeley, Bill Joy created and released the C shell (`csh`) for BSD Unix. The C shell was based on the Mashey shell. It gave existing constructs a more C-like syntax and improved the interactive features of the shell - adding job control and command history. The C shell was succeeded by the Tenex C shell

(tcsh) which further improved the shell’s interactive features - adding command-line editing and completion. The Tenex C shell is the default interactive shell on the FreeBSD operating system.

In 1983, David Korn released the KornShell (ksh). The KornShell is upwardly compatible with the Bourne shell. It introduced associative arrays and floating point arithmetic, while including job control, command history, and command editing features popularized by the C shell and Tenex C shell. The KornShell was designed as a “complete, powerful, high-level programming language” [22]. The KornShell is the default shell on most commercial versions of Unix.

Due to the KornShell’s licensing restrictions a number of open source work-alikes were created. In 1987, Brian Fox created the Bourne-again shell [47] an upwardly compatible and freely available version of the Bourne shell for the Free Software Foundation’s GNU project. The Bourne-again shell incorporates many features from the KornShell and C shell and is the default shell on most Linux distributions. In 1990, Paul Falstad released the Z shell [52]. The Z shell is also upwardly compatible with the Bourne shell and incorporates many of the features from the Bourne-again shell, the KornShell and the Tenex C shell.

In the mid 80s work began at Bell Labs on Plan 9 the successor to Unix. Part of this work was the design of a new shell by Tom Duff. In 1989, the *rc* shell was released for Plan 9 and version 10 of the Unix operating system. Tom Duff described *rc* as similar in spirit but different in detail from Bourne’s shell [9] but this statement could just as easily have been applied to the C shell or Tenex C shell or to the Bourne shell when comparing it to the Mashey or Thompson shell. The details differ among the various Unix shells but there is a strong family resemblance.

The following characteristics are common to most shells (adapted from [7]):

1. Concise syntax
2. Reasonable performance¹

¹The shell is an interactive interface. It must have an acceptable interactive performance so as not to frustrate its users. A shell will also be invoked to interpret scripts and so the

3. The ability to invoke external programs
4. The ability to launch background processes
5. The ability to create and run scripts - text files containing a sequence of commands
6. Filename expansion
7. I/O redirection
8. Pipes
9. Path searching
10. Support for environment variables
11. Control structures, loops, and conditionals
12. Job control
13. Command history and editing
14. Completion

Many of these characteristics were already present in the initial release of the Thompson shell, the first Unix shell. By the time the Bourne shell and C shell were released, the first 11 characteristics would have been considered standard for a Unix shell. All of these characteristics would now be considered standard for a modern Unix shell.

start up time for the shell must be acceptable. This early design goal led to the Bourne shell's unorthodox memory management technique of trapping memory faults in order to match the speed of the Mashey shell.

2.2 S-Expressions

The authors of the *es* shell lamented that a Lisp-like macro system would have, in their opinion, lead to a much more pleasing design for *es* but that macro systems designed for Lisp-like languages do “not mesh well with the free-form syntax that has evolved for UNIX shells” [15].

The Lisp programming language introduced “a fully parenthesized prefix notation for programs and (other) data” [19], referred to as s-expressions. This uniform treatment of code and data facilitates metaprogramming and the implementation of macros by allowing programs to easily process other programs, or program fragments, as data. An s-expression can be defined as follows [25]:

1. Atomic symbols are s-expressions (e.g., X, 42, “Apple Pie”).
2. if e_1 and e_2 are s-expressions, so is $(e_1 . e_2)$.

The atomic symbol *nil* is used to terminate lists which are represented as:

```
(a . (b . (c . (d . nil))))
```

and can be abbreviated as:

```
(a b c d)
```

2.3 The Π -Calculus

Choc’s concurrent core is based on a variant of the π -calculus. The π -calculus is a concurrent model of computation [27]. It builds on the earlier Calculus of Communicating Systems (CCS) and can be seen as the representative for one evolutionary line of the family of concurrent models known as process calculi.

Central to the π -calculus are two concepts. The first is the concept of a channel. Channels are communication and synchronization devices. The π -calculus defines the following channel-related actions:

$$\begin{array}{lll}
P ::= & \text{new } x \, P & \text{restriction} \\
& | & x(y).P \quad \text{input} \\
& | & \bar{x}(y).P \quad \text{output}
\end{array}$$

The expression $\text{new } x \, P$ creates a new channel called x and restricts its visibility to the process P . The expression $x(y).P$ receives a channel along x and binds this to y before continuing as P . The expression $\bar{x}(y).P$ sends the channel y via the channel x before continuing as P .

The second concept is that of a process. The π -calculus adds the following process expressions to the channel-related actions shown above:

$$\begin{array}{lll}
P ::= & P_1 \mid P_2 & \text{concurrency} \\
& | & P_1 + P_2 \quad \text{choice} \\
& | & !P \quad \text{replication} \\
& | & 0 \quad \text{null process} \\
& | & (P) \quad \text{grouping}
\end{array}$$

In the expression $P_1 \mid P_2$, the processes P_1 and P_2 execute concurrently. When presented with the expression $P_1 + P_2$, a non-deterministic choice is made between P_1 and P_2 . Choice is also known as summation. The expression $!P$ means that there are infinitely many copies of the process P . The null process 0 does nothing.

The π -calculus has a number of related subcalculi. The asynchronous π -calculus, for example, adds the restriction that the only process expression that may follow a send operation is the null process. Some subcalculi add a match operator to the set of name actions. Earlier versions of the π -calculus used parameterized processes instead of replication. Other variants add restrictions to replication and choice or even eliminate these altogether.

Translating some familiar Unix idioms to the π -calculus is fairly straightforward. A forking server can be described by a replicated input expression. The choice operator, particularly when restricted to input expressions, strongly resembles the select system call. Unix shells even use the same notation to express

concurrency ('|') with the addition that the standard output of the first process is connected to the next's standard input. The translations for other constructs, however, are less clear. The Unix shell gives the user more direct control over the creation of processes and allows sequential composition as a primitive operation.

2.4 Prototype-Based Programming

Choc's first-class environments enable a flexible form of object-oriented programming commonly referred to as prototype-based, or object-based, programming.

To many, the term object-oriented programming implies language mechanisms for classes and inheritance. There is, however, an alternative view which dispenses with classes and inheritance and replaces these mechanisms with prototypes and delegation. This prototype-based, or as it is sometimes called, object-based, model is considerably simpler than the popular class-based model while being more flexible, more powerful and better suited "to the development of evolving, exploratory and distributed software systems" [41] - characteristics that this style of programming shares with the Unix shell.

"The main features of class-based languages appeared fully formed in Simula. Object-based languages, in contrast, have emerged more gradually" [1]. The common themes in the various prototype-based systems were first stated in [40]. In [8], the authors develop a taxonomy of these variations. We present an adapted form of this taxonomy below.

1. Do objects differentiate between methods and variables? Or, are the members of an object treated as interchangeable slots? In familiar class-based languages like C++ or Java there is a clear distinction between methods and variables. Many prototype-based languages do away with this distinction. Members of an object are treated in a uniform fashion and an object becomes simply a collection of named slots.
2. How are new objects created? Can objects be created ex-nihilo? If so, can objects be created with an initial structure? Prototype-based languages

allow new objects to be created by cloning existing objects. The question is can objects be created from nothing, “ex-nihilo”, or are all objects created by cloning some existing object. If objects can be created “ex-nihilo”, can an initial structure be specified?

3. Can an object’s structure be modified dynamically? Once an object has been created, can its structure be modified? It would be possible to create a language where it is not possible to dynamically modify an object’s structure and where objects can only be created by cloning an existing object but the resulting language would not be very interesting.
4. Is delegation implicit or explicit? Instead of inheritance, prototype-based systems rely on delegation. Often this delegation is implicit. Clones delegate to their parent object behaviour they do not understand. This delegation can be handled by an implicit parent link or at the other end of the spectrum specified on a method-by-method basis.

We will revisit this when describing the style of prototype-based programming Choc enables by exposing environments as first-class values.

Chapter 3

Related Work

3.1 Previous Attempts to Build a Better Shell

Previous attempts to build a better shell “can be divided into two categories: Those that improve the shell as a programming language and those that improve the shell as a command interpreter” [21]. Current Unix shells fall, mainly, in the latter category and have “focused on improving the interactive environment without changing the structure of the underlying language” [15].

Many attempts to improve the shell as a programming language underestimate the importance of a simple syntax suitable for interactive tasks and support, at the syntactic level, for filename expansion, background processes, pipes and redirection [26, 44]. The shell is a language that can be used interactively to chain together commands. This “interactive ‘glue’ aspect of the shell is its key desirable feature” [39]. Attempts to improve the shell at the expense of its minimal syntax must concede that “when executing simple commands the standard Unix command interpreters are preferred” [26]. But, this concession is the same as the admission that the ‘improved’ shell is really only an improved language, as the shell’s power and promise is as a unified programming **and** command language.

Attempts to build a better shell by adding shell-like features to an existing language may result in an improved programming language but typically lack equivalent support for filename expansion, background processes, pipes and/or redirection. The Unix shell’s constraints as a command language have lead to design choices shared with few other languages. As a result, adding shell-like features to an existing language inevitably results in an awkward interactive command language [45]. Attempts to make this type of combination more usable from the command-line result in strange special cases and non-orthogonal features.

The *es* shell and Tcl are two improved shells that, surprisingly, did not displace the Bourne shell and its descendants as the dominant shells on Unix and Unix-like systems.

In 1990, Byron Rakitzis and Paul Haahr started work on the *es* shell [15]. The *es* shell uses the *rc* shell’s syntax while adding lexically scoped variables, first-class functions, rich return values and exceptions. It is surprising that the *es* shell was not more successful. One reason for this may be that “*es* contains features which do not interact well with UNIX itself” [15]. Some of these problems are caused by the interaction (or more accurately lack of interaction) between a subshell and its parent. Exceptions and lexically scoped variables in *es* do not interact well with subshells. We demonstrate a related problem later in this thesis that causes surprising results when using pipes in *es* and other shells. To address these types of problems, Choc was implemented as a concurrent language. This allows Choc to produce less surprising results at the expense of a slightly more complicated implementation. In addition, Choc separates its own internal variables from environment variables. Environment variables are not flexible enough to be used as general purpose variables and so Choc, like Perl, Python and Ruby, separates variables and environment variables.

The *es* shell also makes some confusing design choices. It tries to unify the semantics of passing lists to shell functions and external programs and in doing so restricts lists to being non-hierarchical. At the same time, *es* allows shell functions to return ‘rich’ return values. Choc’s t-expression representation helps in providing more uniform semantics when passing arguments to internal methods and external programs. Each argument to an external program is serialized as a t-expression. This allows sublists to be reconstructed. Choc also defines a status type. The status type is an integer that when used in a boolean context is interpreted as true if its integer value is zero, and false otherwise. This allows Choc to treat the return value from internal methods and external programs in a uniform fashion. At the same time, there is no expectation that a Choc script could return anything other than a small integer value - this is a limitation imposed by Unix.

Tcl could have been a better Unix shell but its focus on simplicity, and constraints as an embeddable language have unfortunately limited its role as an improved shell. Tcl also suffers from some fairly obvious omissions as a command language, such as support at the syntactic level for filename expansion. Tcl has carved out its niche but not as a better Unix shell.

With the release of the PowerShell (formerly Monad shell), Microsoft has developed a capable, and much needed, command-line interface [29]. Before the release of the Monad shell, Windows users wishing to use the command-line could either use the limited built-in command-line interface or install one of many ported Unix shells. Although different in detail, Windows' built-in command-line interface and the PowerShell are both very similar to the Unix shell. Where the PowerShell differs from the Unix shell is in its treatment of pipes. Pipes in the PowerShell pass .NET objects. Whether this is an improvement is still the subject of some debate [46].

3.2 Scripting Languages

“A newer generation of scripting languages has been supplanting sh in Unix” [39]. But, while languages like Perl, Python and Ruby offer many advantages when writing scripts they are still lacking features that are required to be suitable as interactive command languages. There have been a number of attempts to graft shell-like features onto these scripting languages but these have met with limited success [17, 31, 36, 44, 45].

These combinations run into the same problems as other attempts to create a shell based on an existing language - the result is an “uglier, and confusing, language” [45]. What causes problems when grafting shell-like features onto an existing language are the Unix shell's numerous syntactic conventions. When “manipulating Unix processes, nothing beats the terseness and clarity of the [Unix] Shell” [16]. Conventions have evolved over the Unix shell's 40 year history for the use of many standard ASCII characters and these conflict with uses in most other languages. When adding shell-like features to an existing language there are two

choices:

1. Provide an alternate, most likely more cumbersome, and at the very least, unconventional syntax for background processes, filename expansion, pipes and redirection, or,
2. Allow the user to shift between two modes. One which feels more shell-like and allows conventional shell syntax and one which feels more like the host language.

In *rush*, a Ruby shell, there is no concept of current working directory. “Instead, dirs are assigned to variables. There are two variables defined automatically: *root*, and *home*” [44]. When using *rush*, a user performing a simple operation, like renaming a file in his or her home directory, might type something like this:

```
home['original.name'].rename 'new.name'
```

In a standard Unix shell the same command would look like this:

```
mv original.name new.name
```

The price for the features offered by *rush* is a syntax that is significantly more verbose than a typical Unix shell when performing simple tasks.

When a command is interpreted by Zoidberg, a Perl shell, it “undergoes a bit of source filtering to make it play better with the shell environment” [18]. As part of this filtering, Zoidberg identifies potential context blocks and then applies a set of rules to each block to determine if the block should be executed in a shell context or in a Perl context. Some features, like filename expansion, are only available when a command is evaluated in a shell context. The command *mode* can be used to force a particular context to be used. The resulting language inherits the complexity of both languages while adding additional complexity as a result of its modal nature.

3.3 Lisp and the Unix Shell

Lisp and the Unix shell are both dynamic, interactive, interpreted languages. These and other similarities have lead to previous attempts to combine the two [12, 39]. From the right distance the Unix shell's syntax looks very similar to Lisp's. In Lisp we might have:

```
(Command Arg1 ... ArgN)
```

Whereas with the shell we would have:

```
Command Arg1 ... ArgN
```

In the Lisp shell, an early attempt to combine Lisp and the Unix shell, the command scanner even assumed the outer level parentheses which was “a welcome relief from Lots of Insignificant and Stupid Parentheses” [12]. The Lisp shell's command scanner also allowed the backslash character ‘\’ to be used as a continuation character for commands that were too long and needed to be split across multiple lines. (For multi-line statements some sort of continuation character is necessary as parsing must proceed one line at a time).

A more recent attempt to combine Lisp and the Unix shell, is the Scheme shell. Scheme differs from other dialects of Lisp in that variables have lexical scope. Scheme is also properly tail-recursive allowing iterative computations expressed recursively to execute in constant space. In addition, Scheme supports more esoteric programming languages features like hygienic macros and first-class continuations - from which other control structures can be synthesized.

The Scheme shell embeds a Unix shell in the Scheme dialect of Lisp. A more fitting name, however, might be Scheme Script, as in the author's own words:

I wanted a programming language, not a command language, and I was unwilling to compromise the quality of the programming language to make it a better command language. I was not trying to replace use of the shell as an interactive command language. I was trying to

provide a better alternative for writing shell scripts. So I did not focus on issues that might be important for a command language, such as job control, command history, or command-line editing. There are no write-only notational conveniences. I made no effort to hide the base Scheme syntax, even though an interactive user might find all the necessary parentheses irritating [39].

The problem with these attempts to combine Lisp and the Unix shell is that the combination cannot be a simple aggregation, or embedding; decisions have been made in the two worlds that are fundamentally incompatible. As an example, consider the single dot `'.'` which is “both a fundamental Unix file name and a deep primitive syntactic token in Scheme” [39]. Choc’s design was guided by the belief that a successful combination is possible but that the result will be a new language.

Choc borrows heavily from the Scheme dialect of Lisp. The cons-cell is central to its design and many operations in Scheme can be represented directly in Choc. The Choc shell supports lexically scoped variables, is properly tail-recursive and supports functions (referred to as methods in Choc) as first-class values.

Choc is not an attempt to embed a Unix shell in Scheme. Choc replaces s-expressions with t-expressions, is based on the π -calculus rather than the λ -calculus, and adds a built-in prototype-based object system.

3.4 Pict: A Programming Language based on the Π -Calculus

Pict[33] is an experimental language that asks the question, “What would a language based on the π -calculus look like?” It is a fairly direct encoding of an asynchronous choice-free fragment of the π -calculus and one of the few languages based on the π -calculus. (The π -calculus has been used as the basis for modeling languages but rarely as the basis for a programming language). Choc, which is also based on the π -calculus, follows Pict’s lead in design choices like removing summation, restricting replication and including boolean, integer and string types in the core language.

Instead of the mathematical π -calculus notation, Pict uses the following ASCII syntax:

π -calculus	Pict	
$\bar{x}\langle y \rangle.0$	$x!y$	<i>asynchronous output</i>
$x(y).P$	$x?y = P$	<i>input</i>
$P_1 \mid P_2$	$(P_1 \mid P_2)$	<i>concurrency</i>
$\text{new } x \ P$	$(\text{new } x \ P)$	<i>restriction</i>
$!x(y).P$	$x?*y = P$	<i>replicated input</i>

An encoding of the boolean value **false** in Pict is shown below [32]:

```
new b:^[^[] ^[]]

run {- The process "false" -}
  b?[t f] = f![]

run {- The testing process -}
  (new t:^[^[]
    new f:^[^[]
    ( b ![t f]
      | t?[] = print!"It's true"
      | f?[] = print!"It's false" ))
```

Listing 3.1: Pict

Pict's syntax is straight-forward but unfortunately unsuitable as the syntax for a Unix shell. As with the problems encountered when attempting to combine Lisp, or other languages, with the shell, there are fundamental syntactic incompatibilities. This is perhaps not very surprising as Pict's primary motivation was to be a fairly direct encoding of the π -calculus and not the foundation for an interactive command language. We use Pict as a guide rather than trying to embed a Unix shell in Pict.

3.5 SELF and JavaScript

Choc is a prototype-based programming language. Its design was guided by other prototype-based languages. The canonical prototype-based language is SELF [43, 2]. An object in SELF is a collection of slots. A shallow copy of an object can be created by passing the object a `copy` message. An object in SELF can have one or more parent objects. Messages that are not understood are delegated to a parent object. This implicit delegation allows shared behaviour, or traits, to be collected in parent objects. Parent objects play a role similar to that of classes in a class-based language. Slots can be updated or added dynamically.

While SELF is the most often cited example of the prototype-based paradigm, perhaps the most well-known prototype-based programming language is JavaScript [50]. In JavaScript, objects can be created using an initializer which specifies a set of name-value pairs or with a constructor function along with the keyword `new` [50]:

```
// Object initializer.
var obj = { property_1:  value_1,    // property_# may be an identifier...
            2:          value_2,    // or a number...
            ...,
            "property_n": value_n }; // or a string

// Constructor function.
function person(name, age, sex) {
    this.name = name;
    this.age  = age;
    this.sex  = sex;
}

rand = new person("Rand McKinnon", 33, "M");
ken  = new person("Ken Jones", 39, "M");
```

Listing 3.2: Creating Objects in JavaScript

Like SELF, objects in JavaScript are a collection of slots, or properties. Properties can be added, deleted and updated. Cloning is not built-in but can be emulated with various degrees of success [5]. Delegation is implicit and accomplished by setting an object's `__proto__` property.

Chapter 4

Choc

Choc introduces a unified representation for data and programs, similar to Lisp's s-expressions and incorporates many features from the Scheme dialect of Lisp, including first-class functions and lexically scoped variables.

Like the Ada shell, Choc is one of the few attempts to recognize the shell as a concurrent language [6]. (From its description [6], the Ada shell seems to exhibit the same flaws as other attempts to create a shell based on an existing language. At the time of this writing, it was not available in any form that could be evaluated). Choc's design was based on a variant of the π -calculus, also introduced in this thesis, which includes sequential composition as a primitive operation. This variant is used to provide a solid theoretical foundation for Choc and the Choc shell.

Choc also introduces a novel form of first-class environment. These first-class environments enable Choc to support a prototype-based style of object-oriented programming and allow Choc to unify the treatment of objects and modules while simplifying the use and implementation of channels.

4.1 Syntax

The design of Choc started with its syntax. To some, this may seem like a strange place to start. Syntax is often regarded as a trivial implementation detail for a programming language but, when designing a shell, syntax is one of the most important features. The requirement that the shell have a concise syntax is the driving force behind features that the shell shares with few other languages. For example, in most programming languages strings are quoted but the Unix shell allows strings containing no unescaped spaces or other special characters to be

represented directly.

The Choc shell respects the syntax of current Unix shell's, using the same syntax for background processes, filename expansion, I/O redirection and pipes, while generalizing and extending this syntax in the form of t-expressions. T-expressions are fundamental in allowing channels to be implemented as pipes, which allows for better integration with Unix and existing Unix tools. When using the Choc shell, the following commands behave as expected [9]:

```
date
cat /lib/news/build
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
vc *.c &
mk && v.out /*/bin/fb/*
rm -r junk || echo rm failed!
```

4.1.1 T-Expressions

One of Choc's design goals was a shell-like syntax which could be parsed on-line and used to represent semi-structured data. Choc looked to Lisp and the Lisp shell for inspiration.

Following the Lisp shell's lead Choc could have represented an if-statement as shown below. (We choose the if-statement for our running example as the else clause presents special challenges as will be seen later. Instead of an if-statement we could have chosen any other multi-clause statement).

```
if expression \
  (command1) \
```

```
(command2)
```

But the mandatory use of a continuation character for every line in a construct that will most likely span lines seems excessive.

Like Tcl, we could have used curly braces to delimit a sub-block [30]:

```
if expression {
    command1
    command2
}
```

However, this amounts to enforcing the K&R brace style. In the author's experience, it is surprising how passionate programmers can be over a detail as small as brace style and it would be unfortunate to limit the appeal of Choc with such an easily avoidable design choice.

In Choc we decided to use the colon, in addition to the backslash, as a continuation character, which results in the following:

```
if expression:
    (command1) \
    (command2)
```

Given that we indent both `command1` and `command2`, the parentheses are redundant and can be eliminated along with the remaining continuation character by making indentation significant.

```
if expression:
    command1
    command2
```

But in this example how would we represent an else clause (or any similar clause in a multi-clause command)? We would like to avoid a construct like the *rc* shell's 'if not' statement [9]. Ideally, we would like to be able to write something like:

```

if expression:
    command1
    command2
else:
    command3
    command4

```

This can be done if a dedent is interpreted as continuing the original list but now we need some way of indicating that we have reached the end of a multi-line list. For this Choc uses the keyword `end`. So, a multi-line command, like our if-statement, looks like this:

```

if expression:
    command1
    command2
else:
    command3
    command4
end

```

When there is only one sub-command we allow it to follow on the same line as the colon:

```

if expression: command

```

We retain the backslash character as an escape and continuation character (escaping the following newline) and allow curly braces to be used for sublists. For lack of a better name, we refer to this notation as t-expressions. (See appendix A for a more formal definition of t-expressions). The use of t-expressions gives Choc the innate ability to represent semi-structured data and leaves open the possibility of adding a powerful macro system (a possibility that has yet to be explored). In addition, the minimal syntax used by t-expressions ensures that there is little conflict with the Unix shell's numerous existing syntactic conventions.

This representation also allows Choc to easily serialize and transmit information as byte-streams which can, in turn, be easily deserialized, addressing the common criticism that pipes in Unix and the Unix shell restrict communication to the lowest common denominator - unstructured byte streams.

While very similar to Python in the use of indentation and ':' character, Choc is very different in one significant respect. Choc's parser has no concept of an if-statement or a while loop - it only knows about operators and the t-expression syntax used above. Like Lisp, the Choc parser simply constructs lists - it is the Choc evaluator that determines if a particular list is an if-statement. (For more detail on Choc's syntax see appendix A).

The next task in the design of Choc's syntax was to define how a list element, or word, is composed and command operators represented. Conventions have evolved over the Unix shell's 40 year history for the use of many standard ASCII characters. Choc does its best to respect these traditions.

4.1.2 Interactive and Historical Constraints

In addition to alphanumeric characters, the following characters may appear in file or path names or command options. A Unix shell should not use these as special characters so that they may appear unquoted:

- + Used in options. See the Unix 'tail' command
- , Used in filenames (CVS, RCS)
- Used in options and filenames
- . Used in filenames
- / Pathname separator
- _ Used in filenames

The following characters are known as glob characters and may expand to one or more actual filenames:

- * Matches any sequence of characters
- ? Matches exactly one character


```
[ Start of a character class
] End of a character class
~ Used to indicate a home directory
```

The following characters are metacharacters used by most Unix shells:

```
! History substitution
" String delimiter - with substitution
# Comment. (Continues until newline)
$ Variable substitution
& Run command concurrently
' String delimiter - without substitution
( Start group of commands in subshell
) End group of commands in a subshell
; Command separator
< Redirect input
= Assignment operator
> Redirect output
\ Escape character
` Command substitution
{ Start group of commands
| Pipe
} End group of commands
```

The only characters that remain unused from the standard ASCII character set are '%', ':', '@', and '^'. Even these characters are put to use by some shells.

Choc defines the following special characters. These characters must be quoted if they appear in a filename or command option.

```
! Logical negation
" String delimiter
# Comment. (Continues until newline)
```

- % Interpolation operator
- & Run command concurrently
- ' Stop evaluation of expression
- (Start sub-expression
-) End sub-expression
- : Start sub-clause
- ; Sequence operator
- < Redirect input
- = Assignment operator
- > Redirect output
- @ Splice operator
- \ Escape character
- ` Used in command substitution
- { Start sub-list
- | Pipe
- } End sub-list

The Choc shell puts the characters '%', ':', and '@', to use but does not assign any special meaning to the '\$' character. Otherwise, Choc uses the same metacharacters as other Unix shells.

In addition to alphanumeric characters, words in Choc may contain the following characters:

- \$ Variable prefix
- * Matches any sequence of characters (glob)
- + Used in options. See the Unix 'tail' command
- , Used in filenames (CVS, RCS)
- Used in options and filenames
- . Used in filenames
- / Pathname separator
- ? Matches exactly one character (glob)
- [Start of a character class (glob)

```

] End of a character class (glob)
_ Used in filenames
~ Used to indicate a home directory (glob)

```

Words containing glob characters may expand to more than one word depending on the context in which they are used. If this is not the desired effect, glob characters must be escaped with the `\` character or the entire word quoted. The `'$'` character can be used to prefix variable names but this is not required.

Choc treats some words as infix operators. Most of these look like their Unix *test* command equivalents: `-eq`, `-ne`, `-lt`, etc. These operators must be separated from their operands by one or more whitespace characters. (For a more detailed definition of Choc's syntax see appendix A).

4.2 Semantics

4.2.1 Improved Programming Language Features

Many design choices that are clearly unfortunate in retrospect have been carried forward into modern Unix shells. Current Unix shells could be patched to address these problems but not without breaking backward compatibility - resulting in a new shell. As such, the Choc shell revisits these design choices under the assumption that the solution will involve building a new shell that breaks with backward compatibility.

Choc exploits the fact that the programming language features of the shell do not need to be as concise as its interactive features. A successful shell must support what has become the standard syntax for background processes, filename expansion, I/O redirection and pipes. Most users will also expect to be able to use infix notation but the syntax used for defining methods or the fact that variables must be declared has very little impact on the interactive sessions of most users.

In the following discussion, unless otherwise noted, the behaviour of the Bourne-Again shell and the Z shell is identical to that of the KornShell. Some of these

flaws are function-related and so do not apply to the C shell as it and its descendants have an even more serious flaw - a lack of functions altogether.

Explicit Variable Declaration

In the Bourne shell and C shell, variables are not declared. A variable is created with an assignment statement. This means that a typo in the name of a variable on the left hand side of an assignment statement will create a new variable and leave the intended variable's value unchanged. Without a static analysis tool that can highlight variables that are set but never used, this behaviour can be the source of difficult-to-find bugs.

Other languages, like Python, also exhibit this flawed behaviour. Python's `global` statement is a consequence of the ambiguity this behaviour introduces - setting the value of a global variable in a Python function is interpreted as declaring a local variable unless the Python interpreter has been explicitly instructed that this name is global.

In the KornShell, variables can be declared using the `typeset` command but the old behaviour is maintained for backward compatibility making the KornShell just as susceptible to the same types of errors.

```
#!/bin/ksh93

blah() {
    typeset FileName=foo

    # ...

    Filename=bar      # Oops.
}
```

Listing 4.1: Implicit Variable Declaration

All variables in Choc must be declared. Variables are referenced directly - a

sigil character, like the '\$' character used by other shells, is not required. (Choc does allow the '\$' character to appear in variable names. So, although it is not required, it can be used as a prefix).

```
#!/usr/local/bin/choc

define FileName foo

# ...

# Error. 'Filename' is not defined.
Filename = bar
```

Listing 4.2: Explicit Variable Declaration

Choc uses a modified generalized autoquote that was popular in early implementations of Lisp [37]. Arguments to a command are treated as variables, but if no variable is found the value is assumed to be an unquoted string literal. This means that a typo in a variable name can cause a string to be passed where the value of a variable was intended but this is no more error prone (and possibly easier to debug) than the Unix shell's default behaviour of silently inserting the empty string when a non-existent variable is referenced.

We intend to add the ability to turn off this behaviour (when writing larger scripts, for example), in which case all string literals must be explicitly quoted. Scripts where all string literals are quoted form a strict subset of the more permissive behaviour. Similarly when evaluating a command, if the command name is not a built-in operation or user-defined method, it is assumed to be the name of an external command. This is consistent with the behaviour of other shells. We also intend to add the ability to turn off this behaviour, in which case external commands must be invoked with the command `run`.

Better Default Scope

All variables in the Bourne shell are global variables. Consider the following Bourne shell script:

```
#!/bin/sh

blah() {
    foo="I'm a global variable!"
}

blah

echo $foo
```

Listing 4.3: Global Variables

The output of this script is the, rather surprising, text “I’m a global variable!” The KornShell introduced an alternative way to declare functions which along with the command `typeset` which can be used to declare local variables.

```
#!/bin/ksh

function blah {
    typeset foo="I'm a local variable!"
}

blah

echo $foo    # This prints the empty string - $foo is local.

function bleh {
    bar="I'm a global variable!"
}

bleh
```

```

echo $bar

blech() {
    baz="I'm also global variable!"
}
blech
echo $baz

```

Listing 4.4: Local Variables

However, not only is the old behaviour maintained but local variables are only possible when `typeset` is used with the new `function` function declaration. Variables in functions declared using the old function declaration style are still global even if they are declared using `typeset`, as are variables declared without `typeset` in functions declared using the new style.

The `rc` shell only allows a limited form of local variable where an assignment is immediately followed by a command.

This behaviour makes it very difficult to create shell libraries as all functions have the potential to pollute the global name space. Choc does not support the Bourne shell's 'global-by-default' variables.

```

#!/usr/local/bin/choc

define blah: method:
    define foo "I'm a local variable!"
end

blah

# Writes 'foo' as foo is no longer defined.
echo foo

```

Listing 4.5: Local Variables

Safer Dynamic Scope

While not as obvious in the Bourne shell, where all variables are global, in the KornShell, variables referenced in functions declared using the old style have dynamic scope whereas variables referenced in functions using the new style are lexically scoped:

```
#!/bin/ksh93

a='lexical'

foo() {
    echo $a
    a='modified'
}

function bar {
    typeset a='dynamic'

    foo
    echo $a
}

bar
```

Listing 4.6: Dynamic Scope

The danger is that whether a variable has dynamic or lexical scope is not a property of the variable but rather a property of individual functions. Any function with dynamic binding can access and modify variables in its caller's scope.

To support pipes and redirection, a Unix shell needs to support some form of dynamic scope. It is possible for variables in the Choc shell to have dynamic scope

but in Choc, this is a property of the variable rather than a property of the method. This prevents methods from destructively modifying lexically scoped members of their caller's environment. The lexical environment is also searched before the dynamic scope - lexically scoped variables take precedence over dynamically scoped variables.

The KornShell could be improved by breaking backward compatibility. Old style function definitions could be removed and an alternate way to declare variables with dynamic binding provided. This is Choc's approach. Choc retains the fundamental characteristics of a Unix shell but by breaking backward compatibility it is free to re-examine design choices that are clearly unfortunate in retrospect.

```
#!/usr/local/bin/choc

define a "lexical"
dynamic caller "top-level"

define foo: method:
    echo a caller
end

foo

define bar: method:
    define a "local"
    dynamic caller "bar"

    foo
end

bar
```

```

define baz: method:
    dynamic a "nice try"
    dynamic caller "baz"

    foo
end

baz

```

Listing 4.7: Dynamic Scope

The output of the above example is 'lexical' followed by 'modified'. References to the dynamic variable `a` are shadowed by the lexically scoped variable with the same name. The system variables `$stdin`, `$stdout`, and `$stderr` are examples of dynamic variables that are used extensively by Choc. Pipes and redirection would be difficult to implement without this facility.

Richer Data Types

The only data type in early shells is the string. This may seem sufficient given the shell's domain but it becomes obvious fairly quickly that the shell could do with a few additional data types - a list data type in particular.

Later shells like the KornShell added other data types like integers, arrays and associative arrays but, again, maintained old behaviour in the name of backward compatibility.

The shell uses a technique called word splitting to emulate lists using strings of whitespace separated words. (In the Bourne shell and its descendants the IFS environment variable can be set to override the default whitespace separator characters). In many cases this technique works acceptably well:

```

#!/bin/sh

for Number in 1 2 3 4

```

```
do
    echo $Number
done
```

Listing 4.8: Word Splitting

But, there are cases that can cause surprising results:

```
#!/bin/ksh93

touch 'Name With Spaces'

for File in `ls`
do
    cat $File
done

rm 'Name With Spaces'
```

Listing 4.9: Word Splitting (2)

In the above example, the file 'Name With Spaces' is interpreted as three separate file names resulting in error messages when the shell attempts to locate the non-existent files 'Name', 'With' and 'Spaces'.

In addition to strings, Choc adds integers, lists and objects. Choc does not perform word splitting. When capturing the output of a command, the output is split on newlines and each line becomes a list element. This allows Choc to avoid the word splitting problems that affect other shells. Choc uses the *rc* shell's backtick syntax which although it requires two extra characters leads to less ambiguous parsing.

```
#!/usr/local/bin/choc
```

```

touch "Name With Spaces"

ls Name*

for '{ls Name*}: method File:
    cat File
end

rm "Name With Spaces"

ls Name*
```

Listing 4.10: Capturing Output

4.2.2 Improved Support for Concurrency

Unix shells are concurrent languages. This concurrency, however, is poorly integrated with the rest of the language. Attempts to build a better shell have largely ignored this concurrent aspect and instead focused on incorporating functional language features, integrating stronger type checking and/or adding shell-like features to an existing language.

The use of pipes, in particular, can cause unexpected results. The example below produces the surprising result '0' when using the Bourne shell as the Bourne shell forks off a new process for both the left and right-hand side of a pipeline. Pipelines are implemented by the Unix shell as chains of concurrent processes but the Unix shell, as a programming language, has very little concept of concurrency.

```
#!/bin/sh

Count=0

# Goofy way to count lines in this file.
```

```
# This does not work when using the Bourne or Bourne-Again shell.
cat $0 |
while read Line
do
    Count='expr $Count + 1'
done

echo $Count
```

Listing 4.11: Pipes

By default, the Bourne-Again shell maintains this behaviour. The KornShell and Z shell execute the left hand side of a pipe in a subshell which results in the correct output for the above example but scripts can easily be constructed which produce incorrect results when using these shells. The script below produces incorrect results using any shell in the Bourne shell family.

```
#!/bin/sh

Count=0

# Goofy way to show occurrences of 'Count' and count lines
# in this file. This does not work.
cat $0 |
while read Line
do
    Count='expr $Count + 1'
    echo $Line
done |
grep 'Count'
```

```
echo $Count
```

Listing 4.12: Pipes (2)

The C, Tenex C and *rc* shells all exhibit similar behaviour albeit with a slightly different syntax.

In addition to pipelines being implemented as chains of concurrent processes, the Unix shell allows the user to explicitly specify that a task should run 'in the background' with the `&` command terminator. The special variable `$!` is set to the process ID of the last background process and the command `wait` can be used to wait for a list of background processes or all background processes if no process IDs are specified.

Choc is a concurrent language and while concurrent modification of global variables is not encouraged, it is possible, enabling scripts like the one below to produce less surprising results.

```
#!/usr/local/bin/choc

define Count: integer 0
define Line true

# Goofy way to show occurrences of 'Count' and
# number of lines in this file. This works.
cat $0 |
while Line:
    Line =: readline
    Count = Count + 1
    echo Line
end |
grep "Count"
```

```
echo Count
```

Listing 4.13: Pipes

Choc's channel-based approach to concurrency allows particularly elegant solutions to some problems, as in the prime sieve example, adapted from [34], shown below. In this example each time a new prime number is found a process is spawned to filter out multiples of that prime number. These processes are strung together like beads on a string.

```
#!/usr/local/bin/choc

define prime-numbers: channel

define counter: method number:
  define your-welcome: channel

  while true:
    echo number
    echo your-welcome

    your-welcome::read

    number = number + 1
  end
end

define filter: method base:
  define your-welcome: channel

  define number: read
  define thank-you: read
```

```

while true:
    if number -mod base:
        echo number
        echo your-welcome

        your-welcome::read
    end

    thank-you::write

    number =: read
    thank-you =: read
end

end

spawn:
    counter 2 | block:
        define in $stdin

        while true:
            define prime: in::read
            define thank-you: in::read

            echo prime

            define out: channel

            block:
                filter prime &

```



```

        end <in >out

        thank-you::write

        in = out
    end
end
end >prime-numbers

define count: integer 100
while count:
    echo: read
    count = count - 1
end <prime-numbers

```

Choc's concurrent core is based on a variant of the π -calculus. As in [11], we start with the finitary π -calculus, that is the π -calculus without choice, recursion or replication. Because it lacks both replication and recursion the finitary π -calculus is not turing-complete.

We make the following modifications to the finitary π -calculus:

- Replace $P_1 \mid P_2$ with *spawn* $P_1.P_2$, which spawns the process P_1 and then continues on as P_2 . In other words, *spawn* $P_1.P_2 \Rightarrow P_1 \mid P_2$.
- Add *halt*. P . Similar to the way in which the execution of *spawn* results in concurrent processes, the execution of *halt* results in the null process. *halt*. $P \Rightarrow 0$. The process P will never be executed.
- Rewrite *new* x P as *new* x . P . Create the channel x then continue on as P . The visibility of the channel x is restricted to P .
- Rewrite (P) as $(P_1).P_2$.

Our set of operations now looks like this:

$$\begin{array}{lll}
 P ::= & \text{new } x.P & \text{restriction} \\
 | & x(y).P & \text{input} \\
 | & \bar{x}(y).P & \text{output} \\
 | & \text{spawn } P_1.P_2 & \text{concurrency} \\
 | & \text{halt}.P & \text{termination} \\
 | & (P_1).P_2 & \text{grouping}
 \end{array}$$

If an action is followed by the null process we allow the $.0$ to be omitted. Sequential composition is implicit in all of these operations. Factoring out sequential composition we have the following set of operations:

$$\begin{array}{lll}
 P ::= & \text{new } x & \text{restriction} \\
 | & x(y) & \text{input} \\
 | & \bar{x}(y) & \text{output} \\
 | & \text{spawn } P & \text{concurrency} \\
 | & \text{halt} & \text{termination} \\
 | & (P) & \text{grouping} \\
 | & P_1; P_2 & \text{sequence}
 \end{array}$$

By adding a loop construct we can express guarded replication and we have arrived at the turing-complete variant of the π -calculus we use as the basis for Choc. To summarize, we define the following operations:

$$\begin{array}{lll}
 P ::= & \text{new } x & \text{restriction} \\
 | & x(y) & \text{input} \\
 | & \bar{x}(y) & \text{output} \\
 | & \text{spawn } P & \text{concurrency} \\
 | & \text{halt} & \text{termination} \\
 | & (P) & \text{grouping} \\
 | & P_1; P_2 & \text{sequence} \\
 | & \text{loop } P & \text{repetition}
 \end{array}$$

Choc is defined as a series of extensions to this core language. We follow the lead of languages like Pict and add the primitive types: boolean, float, integer and string to our core language. For efficiency, the implementation of Choc represents these directly rather than encoding these as channels [33]. We also add various operations for these types. As we’ve added booleans, we extend our core language with the familiar if/else statement and replace loop with the more general while-statement. (For Choc’s full language definition see appendix A).

4.2.3 Improved Support for Abstraction

One thing missing from the variant of the π -calculus used as the basis for Choc, is a mechanism for abstraction. We add operations to support the definition and application of functions. In the π -calculus, functions can be implemented by a replicated input expression that establishes a private communication channel. The replicated input expression accepts a communication channel when the function is invoked which is used to pass the function arguments and a return channel. We add operations to support this directly. Adding this is very similar to adding parameterized processes, except that parameterized processes do not return a value. Functions are more general. If a parameterized process is desired, the return value can simply be ignored.

For consistency with these new function operations we now define all operations as ‘returning a value’. We separate the naming and creation of a channel (names may also be applied to functions) and rewrite input and output operations so that they accept and return a value, respectively. A side-effect of these changes is that we now have process expressions and value expressions. We add a conversion operation to converting a process expression to a value expression.

$P ::=$	$define\ w\ [e]$	<i>definition</i>
	$channel$	<i>channel creation</i>
	$read\ e_c$	<i>input</i>
	$write\ e_c\ e$	<i>output</i>
	$spawn\ P$	<i>concurrency</i>
	$halt$	<i>termination</i>
	(P)	<i>grouping</i>
	$P_1; P_2$	<i>sequence</i>
	$loop\ P$	<i>repetition</i>
	$function\ [w_1 \dots w_n]P$	<i>abstraction</i>
	$e_f\ e_1 \dots e_n$	<i>application</i>
$e ::=$	w	<i>variable reference</i>
	$\{P\}$	<i>conversion</i>

Where w is a word and e is an expression. (The expressions e_c and e_f evaluate to a channel and a function, respectively).

In current Unix shells, support for abstraction is almost non-existent. The C shell and its descendents lack support for functions altogether.

4.2.4 Improved Support for Modularity

The Choc shell offers much richer support for modular programming than current Unix shells. The closest thing to support for modular programming in current Unix shells is the shell's `source` or `.` command which executes commands read from the specified file in the current environment.

Choc is a prototype-based, or object-based, language [1]. This paradigm is a natural fit with an interactive command language as it supports the immediate use of objects. Objects, in Choc, are implemented by exposing environments as first-class values. Exposing environments as first-class values allows Choc to unify the treatment of objects and modules and simplify the use and implementation of channels.

To create new object in Choc, the `object` command is used. The body of an `object` command is executed and the resulting environment becomes the return value. Variables defined using the command `public` are accessible 'outside' an object. Variables created with the command `define` are private and not accessible outside an object. The `::` operator is used to access public members of an object.

Accessing a member of an object is implemented by evaluating the identifier in the environment represented by the object. In the example below the identifier `m` is evaluated in the environment `o`:

```
o::m    # Evaluate m in the context of o.
```

We can describe the prototype-based features of Choc by returning to the taxonomy for prototype-based languages described earlier.

1. Do objects differentiate between methods and variables? Or, are the members of an object treated as interchangeable slots? Like SELF and JavaScript, objects in Choc are collections of name-value pairs, or slots.
2. How are new objects created? Can objects be created ex-nihilo? If so, can objects be created with an initial structure? Objects in Choc can be created with an initial structure. It is also possible to create clones and descendent objects.
3. Can an object's structure be modified dynamically? Public members may be added dynamically to all objects.
4. Is delegation implicit or explicit? Because objects in Choc are implemented by exposing environments, which maintain a link to their parent environment, all objects have an implicit parent link. In the current implementation this link cannot be modified. All objects in Choc delegate behaviour they do not understand to their parent object using this implicit parent link. Explicit delegation is also possible but must be specified on a method-by-method-basis.

Describing these operations in terms of the primitives from the previous section:

$P ::=$	<i>define</i> $w [e]$	<i>private definition</i>
	<i>public</i> $w [e]$	<i>public definition</i>
	<i>channel</i>	<i>channel creation</i>
	<i>object</i> P	<i>object creation</i>
	<i>spawn</i> P	<i>concurrency</i>
	<i>halt</i>	<i>termination</i>
	(P)	<i>grouping</i>
	$P_1; P_2$	<i>sequence</i>
	<i>loop</i> P	<i>repetition</i>
	<i>method</i> $[n_1 \dots n_n]P$	<i>abstraction</i>
	$e_m e_1 \dots e_n$	<i>application</i>
$e ::=$	w	<i>variable reference</i>
	$e_o :: e$	<i>contextual evaluation</i>
	$\{P\}$	<i>conversion</i>

Converting channels to objects allows the **read** and **write** primitives to be eliminated. These become members of individual channel objects. The methods **clone**, **child** and **public** are members of all objects. The **clone** method creates a shallow copy of an object. The **child** method creates a new object with the receiver as its parent. The **public** method can be used to add new public members to an existing object. As environments are now first-class, we add an operation to evaluate an expression in the context of given environment/object.

4.3 Pragmatics

Much of this section was adapted from [20] which describes using the Bourne shell or KornShell. As demonstrated below the Choc shell is very similar to other Unix shells for everyday use. Areas where the behaviour of the Choc shell differs are

noted and explained. The '>' prompt is used to indicate interaction with the Choc shell whereas the '\$' prompt is used to indicate the Bourne shell or KornShell.

4.3.1 Command-Line Structure

The simplest command is the single word:

```
> who
you      tty2      Sep 28 07:51
jpl      tty4      Sep 28 08:32
```

A command ends with a newline. A semicolon can be used to sequence commands entered on the same line.

```
> date;
Wed Sep 28 09:07:15 EDT 1983
> date; who
Wed Sep 28 09:07:15 EDT 1983
you      tty2      Sep 28 07:51
jpl      tty4      Sep 28 08:32
```

If we try sending the output of `date; who` through a pipe, as shown below, with a typical Unix shell, only the output of `who` is sent through the pipe - parentheses must be used to group commands in a typical Unix shell, due to the fact that `';`, as a command terminator, has a lower precedence than `'|'`:

```
$ date; who | wc
Wed Sep 28 09:08:48 EDT 1983
      2      10      60

$ (date; who)
Wed Sep 28 09:11:15 EDT 1983
you      tty2      Sep 28 07:51
jpl      tty4      Sep 28 08:32
$ (date; who) | wc
      3      16      89
```

In Choc ';' is a sequence operator not a command terminator. Choc also defines ';' as having a higher precedence than '|'. If we would like to pipe the output of `date`; `who` through a pipe, when using the Choc shell, we can simply type:

```
> date; who | wc
      3      16      89
```

The Choc shell uses parentheses to group expressions. The `block` construct is used to group commands. If we only wanted the output from `who` to be piped to `wc` but we still wanted the commands `date` and `who` executed as a unit we could type:

```
> block:
    date
    who | wc
end
Wed Sep 28 09:12:00 EDT 1983
      2      10      60
```

The `block` command is more verbose than the parentheses that are used to group commands in a typical Unix shell. Choc exploits the fact that treating ';' as a sequence operator and not a command terminator results in a more concise syntax and relies on the availability of the `block` command when the intent is for sequence to have a lower precedence. If Choc did not change how ';' was interpreted the more verbose `block` command would need to be used for all groupings. (Note: this behaviour has not yet been implemented. Currently, ';' is treated as a command terminator).

As with other shells, data flowing through a pipe can be tapped and placed in a file:

```
> date; who | tee save | wc
      3      16      89
```



```
> cat save
Wed Sep 28 09:13:22 EDT 1983
you      tty2      Sep 28 07:51
jpl      tty4      Sep 28 08:32
> wc <save
      3      16      89
```

To execute a long running command 'in the background' we can use the '&' operator:

```
> long-running-command &
```

The Choc shell does not print the process id for the background command. Instead the '&' operator returns a process handle which can be used by commands like `wait`. In other Unix shells '&' is a command separator but in Choc it is a postfix operator that can be applied to a command. In the current implementation there is very little that a process handle can be used for, other than the target for a `wait` command, but we envision extending this in later implementations to provide more convenient interaction with asynchronous processes. The '&' operator along with the `sleep` command provides an easy way to run a command in the future:

```
> sleep 300; echo Tea is ready &
```

Again, with a typical Unix shell the sequence of commands would need to be grouped using parentheses but Choc assigns a higher precedence to ';' so there is no need to explicitly group the commands to achieve the desired effect allowing a slightly more concise syntax.

As with current Unix shells, we can also run a pipeline in the background:

```
> pr file | lpr &
```

4.3.2 Metacharacters

The Choc shell uses standard shell glob characters:

```
> ls
junk
temp
> echo *
junk temp
> echo .*
. .. .profile
```

To stop the Choc shell from interpreting special characters, we can enclose them in double quotes:

```
> echo "***"
***
```

Unlike other shells, Choc does not use the single quote character to delimit strings. Instead it uses the single quote character to stop an expression from being evaluated. This difference is due to the incorporation of Lisp-like features - specifically t-expressions. Choc uses the same representation for data and programs and so, like Lisp, needs a way of specifying that an expression should be treated as data and not evaluated.

Current Unix shells automatically perform string interpolation [4] on double quoted strings but not single quoted strings. Using the single quote to stop an expression from being evaluated creates a problem which Choc solves by never automatically performing string interpolation - Choc does not 'peek inside' quoted strings unless explicitly instructed to do so with the interpolation operator - '%':

```
> echo "%s, %s!" % '{hello world}'
hello, world!
```

This type of string formatting is more verbose but also less prone to parsing ambiguities and more flexible in that format strings can be specified separate from data. (Note: the string interpolation operator has not yet been implemented).

Processing of backslashes is more complicated in Choc than in other shells. A literal backslash can be represented using two backslashes, some ASCII control characters can be represented with a special backslash sequence ('`\n`', '`\r`', '`\t`'), while a backslash followed by a newline is replaced with nothing - the backslash character also acts as a continuation character. When a glob character like '*' or '?' or '[' is preceded by a backslash the sequence is preserved unchanged:

```
> echo \*\*\*
\*\*\*
```

Preserving backslash glob sequences simplifies the application of glob characters in Choc for uses other than filename matching.

String concatenation is not implicit in Choc. To concatenate strings the concatenation operator '^' must be used:

```
> echo "hello, " ^ world ^ !
hello, world!
```

Implicit string concatenation is a source of parsing ambiguity in current Unix shells. (Note: the string concatenation operator has not yet been implemented).

The metacharacter '#' is used by Choc to indicate a comment. A comment starts with the '#' character and continues until the end of the line:

```
> echo hello # there
hello
```

4.3.3 Creating New Commands

Suppose we want to create a command to count the number of users, using the pipeline:

```
> who | wc -l
```

The first step is to create an ordinary file that contains the line `who | wc -l` either with an editor or using the shell:

```
> echo "who | wc -l" >nu
```

As with current Unix shells, the Choc shell is a program we can run and redirect its input:

```
> who
you      tty2      Sep 28 07:51
rhh      tty4      Sep 28 10:02
moh      tty5      Sep 28 09:38
ava      tty6      Sep 28 10:17
> cat nu
who | wc -l
> choc <nu
      4
;
```

We can also execute nu by typing:

```
> choc nu
```

To execute nu directly we must make nu executable:

```
> chmod +x nu
```

As `/bin/sh` is the default interpreter we will also need to add a directive to specify that the program `choc` is the interpreter that should be used to execute nu:

```

block:
    echo "#!/usr/local/bin/choc"
    echo "who | wc -l"
end >nu

```

Once this is done we can execute nu directly:

```
> nu
```

All Choc scripts should start with the line:

```
#!/usr/local/bin/choc
```

Where `/usr/local/bin/choc` is the full path to the Choc interpreter.

4.3.4 Command Arguments and Parameters

Most shell programs interpret arguments, so that, for example, filenames and options can be specified when the program is run.

Arguments to a Choc script are stored in the list `$args` in the same way that variadic parameters are passed to methods. To access the first argument we can use Choc's Lisp-like list manipulation methods:

```
echo: car $args    # Print out the first element of $args.
```

The name `$0` is bound to a name of the program being executed.

Choc does not perform word splitting but a list can be expanded and passed as arguments using the splice operator, `@`.

```
chmod +x @$args.    # Expand the list of arguments.
                   # Given an empty list, expand to nothing.
```

4.3.5 Program Output as Arguments

The output of a program can be captured in a list using the backquote operator. This list can then be expanded using the splice operator:

```
mail @'{cat mailinglist} <letter
```

Newlines are interpreted as word separators and the resulting list is spliced into the list of arguments being passed to the `mail` command. Choc's combination of backquote and splice requires two extra characters over the typical Unix shell's backquotes. The advantage to this notation, in addition to being more regular, is that it can be nested easily. The `rc` shell uses the same notation for backquotes but does not separate capturing output as a list and expanding that list into individual arguments.

4.3.6 Objects

A new object can be created “ex nihilo” with the `object` command. Private members are defined using the command `define` and public members are defined using the command `public`.

```
define point: method r s: object:
  define x: integer r
  define y: integer s

  public get-x: method:
    return self::x
  end

  public get-y: method:
    return self::y
  end
```

```

    public move: method a b:
        self::x += a
        self::y += b
    end
end

define p: point 0 0

```

In the above example, we create a generator method, `point`, which acts very much like a constructor in class-based languages, but we could also use the `clone` method to create a clone of an existing object or the `child` method to create a descendent object. The methods `get-x`, `get-y` and `move` could also be moved up and out to the same level as the definition for `point`. All objects created by the generator method, `point`, maintain an implicit parent link to their enclosing scope and, in doing so, would automatically delegate calls to these methods to their parent object. If, for example, we wanted other objects like circles or squares, we could use methods defined in the enclosing scope to implement shared behaviour.

Dynamic communication patterns are enabled by the fact that public slots can be updated or added to any object, at runtime. This combined with objects that can be defined and used immediately, rather than in a two step template-instance, process makes this style of object-oriented programming more suited to the Unix shell's interactive and exploratory nature.

An object's members can be accessed using Choc's cons operator - `::`. Because of the Choc shell's syntax constraints there are a limited number of characters available for additional operators and more familiar operators for member access, like `'.'` or `'->'`, cause ambiguity when parsing. Using the cons operator for this purpose allows the Choc shell to evaluate a configuration of cons cells that would otherwise be unused and reported as an error. Since objects in Choc can be thought of as first-class name spaces, the use of `::` as both a cons and name space operator also mirrors these uses in other languages.

Only public members are accessible 'outside' an object. For example, to move

the point defined above by 2 units along both the x and y axis, we must do the following:

```
p::move 2 2
```

All Choc objects have a `clone` method that can be used to create a shallow copy of the current object and a `child` method that can be used to create a descendent object.

```
define o: p::clone
```

New public members may be added to any object:

```
o::public print: method:
  write "("
  write self::x
  write ","
  write self::y
  write ")\n"
end
```

Public methods may access private members. This means it is possible to add public methods that exposes private members. Choc follows Python's "we're all adults here" motto.

Choc uses `object` to implement its module system. Choc supports the `'.'` or `source` command, where:

```
. file-name
```

inserts the contents of `'file-name'` as if they were typed directly. The Choc command:

```
import file-name
```

is equivalent to:


```
object:
    . file-name
end
```

and can be used as follows:

```
define module: import file
```

Public, top-level definitions in ‘file’ can now be accessed using ‘module’ and the ‘::’ operator.

Implicit delegation allows Choc to support traits which are implemented as methods defined in the parent environment. Explicit delegation is accomplished by creating a member in one object and assigning it a method in another object. Redirection is also possible and is accomplished by creating a method that calls another object’s method.

```
# Trait. All objects 'inherit' this method.
```

```
public me: method: echo $self::name
```

```
define x: object:
    define name "x"
end
```

```
define y: object:
    define name "y"
    public me-too: x::me    # Explicit delegation.
end
```

```
define z: object:
    define name "z"
    public you: method: x::me    # Redirection.
end
```

Chapter 5

Implementation Details

The Choc shell is implemented in C. The availability of tools like `re2c`, the `lemon` parser generator and libraries like the `tecla` command-line input library, Boehm-Weiser-Demers conservative garbage collection library and POSIX threads library were instrumental in the development of Choc.

5.1 Storage Management

As mentioned in [35], “the management of storage is central to the implementation of any language” and Choc is no exception. Rather than implementing its own garbage collector Choc relies on the Boehm-Demers-Weiser conservative garbage collector. The cell is Choc’s atomic storage unit. In addition to strings, cells in Choc can be booleans, integers, floats, symbols, pairs, methods, objects and channels. Choc also introduces an integer type called `status` that when evaluated in a boolean context returns `true` when its value is 0 and `false` otherwise. This allows Choc to handle the exit status of Unix processes and related aspects of the language in a uniform fashion.

With the exception of boolean and channel cells, cells are allocated in page-sized, and page-aligned, blocks. An integer type identifier is stored once per block. As there are only two boolean values they are individually allocated. Channels are allocated individually so that the underlying pipe can be closed when the channel is collected. With other types, that are not allocated individually, fragmentation is possible. If a block has a single live cell it cannot be reclaimed. One solution to this problem would be to add a copying collector to be invoked when memory is scarce and fragmentation is high. This is not currently implemented.

Choc has no pointer type but the implementation of Choc uses pointers extensively. Choc uses tagged pointers [14] - the size and alignment of types is exploited in order to use the least significant bits in a cell's address for various flags.

5.2 Processes and Channels

Rather than implementing its own threads, Choc relies heavily on POSIX threads. In the current implementation, channels are implemented as a thin wrapper over pipes. Channel writes are serialized using Choc's native t-expression format and channel reads are deserialized. This allows Choc to communicate with (and even through) existing Unix utilities. One drawback to this approach is that it is inefficient when the two communicating entities are actually threads in the same address space.

Buffering changes the semantics of writing to a channel. The current implementation embraces this and treats writes as non-blocking operations. In order to use channels for synchronization, two channels must be used - a communication and acknowledgment channel.

On FreeBSD it is possible to modify the kernel so that pipe writes respect the `O_DIRECT` flag and then set this flag on all pipes used for channels. This then causes direct (blocking) writes to be used even for small transfers. Interleaving reads can also cause problems if two or more Unix processes are sharing a channel as it is possible for the low-level Unix routines to accidentally read across message boundaries. The `O_DIRECT` kernel modification can help here as can limiting the size of messages sent across pipes that may be shared by more than one Unix process. There may be other ways to mitigate these problems both on FreeBSD and operating systems other than FreeBSD but they have not been investigated by the author.

5.3 The Choc Interpreter

The Choc interpreter is loosely based on the evaluator described in [10]. It is written in C but implements its own stack. The stack determines the evaluator's current state. Restore states are implemented as a bitmask. When pushing a restore state on the stack, the stack is first inspected. If the topmost element is a restore state and a subset of the restore state being pushed, the topmost element is overwritten. In this way the stack does not grow unnecessarily - enabling proper tail calls. In addition to the stack, the evaluator maintains a reference to both the current lexical and dynamic environment, the code it is currently evaluating and a scratch variable that functions as a combination accumulator and current value `rib`. All of these are stored on a per process/thread basis.

5.4 Current Limitations

The following are limitations of the current implementation:

- `';` is currently treated as a synonym for a newline and not as a sequence operator.
- Choc's `'strict'` mode is not yet implemented.
- The string concatenation and interpolation operators are not yet implemented.
- Quasiquote and unquote are not yet implemented.
- Escaping of glob characters does not work correctly on FreeBSD due to a broken implementation of `glob`. Choc will most likely transition to `tecla's ef_expand_file`.
- Detection and formatting of recursive structures is not implemented
- Environment variable, job control and signal handling features are missing.
- Error checking is minimal and error messages vague.

Chapter 6

Conclusion

6.1 Results

Choc retains the familiar look and feel of current Unix shells while making considerable improvements to the shell as a programming language. As Choc demonstrates, a combined command and programming language can satisfy the Unix shell's interactive and historical constraints while incorporating advanced programming language features. When using the Choc shell, the following commands behave as expected [9]:

```
date
cat /lib/news/build
who >user.names
who >>user.names
wc <file
echo [a-f]*.c
who | wc
who; date
vc *.c &
mk && v.out /*/bin/fb/*
rm -r junk || echo rm failed!
```

Choc shares the following characteristics with other Unix shells:

1. Concise syntax
2. Reasonable performance

3. The ability to invoke external programs
4. The ability to launch background processes
5. The ability to create and run scripts - text files containing a sequence of commands
6. Filename expansion
7. I/O redirection
8. Pipes
9. Path searching
10. Support for environment variables
11. Control structures, loops, and conditionals
12. Job control¹
13. Command history and editing
14. Completion

By borrowing heavily from the Scheme dialect of Lisp, Choc brings the power of higher-order programming to the shell. The concurrent features of the shells are integrated into the underlying language and extended by introducing channels. In addition to providing an expanded set of built-in data types, Choc's first-class environments enable a flexible form of object-oriented programming that complements the shell's interactive nature. Even in its current embryonic state, the Choc shell compares favorably with other Unix shells (the table below was adapted from [49]):

¹The current implementation is missing features for job control. It is not clear, particularly in the presence of increasingly ubiquitous graphic user interfaces whether these features should still be provided by a Unix shell.

As noted, some features have not yet been implemented. We expect the implementation of these features to be relatively straightforward.

Choc exploits the fact that the programming language features of the shell do not need to be as concise as its interactive features. A successful shell must support what has become the standard syntax for background processes, filename expansion, I/O redirection and pipes. Most users will also expect to be able to use infix notation but the syntax used for defining methods or the fact that variables must be declared has very little impact on the interactive sessions of most users.

Like other Unix shells, Choc was designed as a 'glue' language [39]. It was designed to coexist with other programs, other scripting languages and even other shells. It is not an attempt to compete with languages like Perl, Python or Ruby but rather to push the limits of the Unix shell further into the space occupied by these languages, allowing users the choice to remain in the shell rather than being forced out due to artificial limitations.

6.2 Future Work

The concept of a shell is clearly a powerful one and major improvements in using a system such as Unix can be obtained by increasing the facilities offered by the shell [3].

The Choc shell demonstrates that significant improvements to the programming language features of the Unix shell are possible but there is much more work to be done before the Choc shell can be considered a production-ready shell.

The current implementation is missing features for job control. It is not clear, particularly in the presence of increasingly ubiquitous graphic user interfaces whether these features should still be provided by a Unix shell.

Signal handling is also missing in the current implementation. Signal handling is complicated by the Choc shell's use of threads. The use of threads exposes many areas in Unix that struggle with the changes threads present. One aspect of the implementation that proved surprisingly problematic was the concept of a current working directory. The current working directory is a property of the

Unix process as a whole rather than individual threads. Choc threads are forced to keep track of their current working directory and this must always be set in child processes after forking but before executing an external command to ensure that the external command executes in the correct location.

Choc has not been benchmarked extensively. Informally, its memory usage, start up time and response time in interactive sessions have been found to be comparable with other Unix shells but more work should be devoted to ensuring competitive performance. The command history, command editing and completion facilities are handled by the *tecla* command-line input library. The *tecla* command-line input library enables comfortable interactive use but much more work is required before Choc can compete with these facilities in shells like the Z shell and Bourne-Again shell which offer multi-line editing and customizable completion.

The author's experience with Choc has been limited to small scripts and short interactive sessions. Choc's userbase (the author) is also quite small. Initial results are promising but more experience with Choc is needed.

The Choc shell's t-expression syntax would work well as a dependency language for a **make** front-end or replacement. Adding **expect**-like functionality would enable the scripting of less cooperative applications. Other concurrent languages [34, 38] have been suggested as more natural languages to use in the implementation of window systems and graphic user interfaces. The Choc shell's t-expression syntax would also work well for describing the hierarchical structures that naturally arise when composing widgets. It would be interesting to investigate a window system and applications developed in, and/or extendable using, Choc.

Appendix A

Language Specification

A.1 Words

In addition to alphanumeric characters, words in Choc may contain the following characters:

- \$ Variable prefix
- * Matches any sequence of characters (glob)
- + Used in options. See the Unix 'tail' command
- , Used in filenames (CVS, RCS)
- Used in options and filenames
- . Used in filenames
- / Pathname separator
- ? Matches exactly one character (glob)
- [Start of a character class (glob)
-] End of a character class (glob)
- _ Used in filenames
- ~ Used to indicate a home directory (glob)

A.2 Grammar

Words in Choc are grouped into commands according to the following grammar:

```
command ::= command AMPERSAND
command ::= command C_COE opt_separator expression
command ::= command C_COC opt_separator command
```

```
command ::= statement
command ::= statement END

command ::= statement END statement

statement ::= list
statement ::= list sub_statement
statement ::= sub_statement
statement ::= expression ASSIGNMENT opt_separator rhs

rhs ::= expression
rhs ::= sub_statement

sub_statement ::= COLON statement
sub_statement ::= COLON sub_block statement
sub_statement ::= COLON sub_block

sub_block ::= separator INDENT block separator DEDENT

block ::= command
block ::= block separator command

list ::= expression
list ::= list expression

expression ::= LEFT_BRACE RIGHT_BRACE
expression ::= LEFT_BRACE command RIGHT_BRACE
expression ::= LEFT_PAREN expression RIGHT_PAREN
```

expression ::= SINGLE_QUOTE expression

expression ::= expression DOUBLE_COLON expression

expression ::= expression E_EOE_0 opt_separator expression

...

expression ::= expression E_EOE_F opt_separator expression

expression ::= word

expression ::= QUOTED_STRING

where C_COE is one of:

<	Redirect standard input
>	Redirect standard output
>>	Redirect standard output (append)
!>	Redirect standard error
!>>	Redirect standard error (append)

and C_COC is one of:

&&	Command and
	Command or
	Pipe standard output to standard input
!	Pipe standard error to standard input

and E_EOE_0 through E_EOE_F are:

%	String interpolation
+	Addition
-	Subtraction
^	String concatenation
-and	Boolean and
-div	Division

<code>-eq</code>	Equality
<code>-ge</code>	Greater or equal
<code>-gt</code>	Greater than
<code>-is</code>	Reference (pointer) equality
<code>-le</code>	Less or equal
<code>-lt</code>	Less than
<code>-mod</code>	Modulo
<code>-mul</code>	Multiplication
<code>-ne</code>	Not equal
<code>-or</code>	Boolean or

Note that the `E_EOE_0` through `E_EOE_F` operators shown above must be separated from their operands by spaces.

A.3 Commands

(Many of the following commands are adapted from [19]).

`<method> [<expression1> ... <expressionN>]`

Evaluates `<method>` and then applies this to the arguments specified.

`append <expression1> [... <expressionN>]`

Returns a list consisting of the elements in `<expression1>` followed by the elements of the other lists.

`block <sub_statement>`

Creates a new dynamic environment and lexical scope. Executes the command(s) in `<sub_statement>`. Returns the value returned by the last command in `<sub_statement>`.

`boolean [<expression>]`

Creates a new boolean object. Initial value is the boolean value of `<expression>` or False if `<expression>` is not specified.

`cd <expression>`

Changes the value of the current working directory in the current dynamic environment. Returns the path of the new current working directory.

`channel`

Creates a new channel object.

`car <expression>`

Returns the head of the pair `<expression>`.

`cdr <expression>`

Returns the tail of the pair `<expression>`.

`cons <expression1> <expression2>`

Returns a new pair whose head is `<expression1>` and whose tail is `<expression2>`.

`define <word> [<expression>]`

If the current lexical scope does not contain a private member named `<word>`, creates a new private member named `<word>` in the current lexical scope. Sets the value of the private member `<word>` in the current lexical scope to `<expression>` or Nil if no expression is given.

`double [<expression>]`

Creates a new double object. Initial value is the double value of `<expression>` or 0.0 if `<expression>` is not specified.

`dynamic <word> [<expression>]`

If the current dynamic environment does not contain a member named `<word>`, creates a new entry named `<word>` in the current dynamic environment. Sets the

value of `<word>` in the current dynamic environment to `<expression>` or Nil if no expression is given.

`echo <expression1> [... <expressionN>]`

Writes serialized arguments to standard output. Return value is undefined.

`for <expression1> [... <expressionN>] <method>`

The expressions `<expression1>` through `<expressionN>` must evaluate to lists of the same length. The method `<method>` must be a method that accepts as many arguments as there are lists. The method `<method>` is called element-wise on all the elements in the list(s) specified. Returns a list of the values returned by each call to `<method>`. Except for the ordering of `<expression1>` through `<expressionN>` and `<method>` this command is identical to Scheme's `map`.

`if <expression> <sub_statement> [else <command>]`

Creates a new dynamic environment and lexical scope. If `<expression>` evaluates to True, executes the command(s) in `<sub_statement>`. Otherwise, if an else clause is given, executes `<command>`. Returns the value returned by the last command in `<sub_statement>` or `<command>` depending on which was executed.

`integer [<expression>]`

Creates a new integer object. Initial value is the integer value of `<expression>` or 0 if `<expression>` is not specified.

`is-channel <expression>`

Returns true if `<expression>` is a channel, false otherwise.

`is-double <expression>`

Returns true if `<expression>` is a double, false otherwise.

`is-integer <expression>`

Returns true if <expression> is an integer, false otherwise.

`is-list <expression>`

Returns true if <expression> is a list, false otherwise.

`is-method <expression>`

Returns true if <expression> is a method, false otherwise.

`is-null <expression>`

Returns true if <expression> is the empty list, false otherwise.

`is-pair <expression>`

Returns true if <expression> is a pair, false otherwise.

`is-status <expression>`

Returns true if <expression> is a status, false otherwise.

`is-symbol <expression>`

Returns true if <expression> is a symbol, false otherwise.

`length <expression>`

Returns the length of the list <expression>.

`list <expression1> [... <expressionN>]`

Returns a newly created list of its arguments

`method [<word1> ... <wordN>] <sub_statement>`

Creates and returns a new method object.

`object <sub_statement>`

Creates a new dynamic environment and lexical scope. Executes the command(s) in `<sub_statement>`. Returns a reference to the lexical scope.

`public <word> [<expression>]`

If the current lexical scope does not contain a public member named `<word>`, creates a new public member named `<word>` in the current lexical scope. Sets the value of the public member `<word>` in the current lexical scope to `<expression>` or Nil if no expression is given.

`quote <expression>`

Evaluates to `<expression>`.

`reverse <expression>`

Returns a newly allocated list containing the elements of `<expression>` in reverse order.

`set-car <expression1> <expression2>`

Sets the car of `<expression1>` to `<expression2>`. Returns `<expression1>`.

`set-cdr <expression1> <expression2>`

Sets the cdr of `<expression1>` to `<expression2>`. Returns `<expression1>`.

`spawn <sub_statement>`

Spawns a new thread to execute the command(s) in `<sub_statement>`. Returns a process handle object that can be passed to the `wait` command.

`status [<expression>]`

Creates a new status object. Initial value is the status value of `<expression>` or False if `<expression>` is not specified.

`wait <expression1> [... <expressionN>]`

Waits for the specified threads to complete. Return value is the value returned by the last thread.

`while <expression> <sub_statement>`

Creates a new dynamic environment and lexical scope. Executes the command(s) in `<sub_statement>` while `<expression>` evaluates to True. Return value is undefined.

Appendix B

Source Code

The following is a snapshot of the Choc source taken on May 12, 2010. The current Choc source can be found at <http://www.scs.carleton.ca/~mmacinn2/choc>.

B.1 License

This software is provided under the terms of the MIT-style license below:

Copyright (c) 2009–2010 Michael MacInnis

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Listing B.1: license.txt

B.2 Common

```

/* See license.txt for copyright and licensing details. */

#ifndef CommonH
#define CommonH

#include "config.h"

#define GC_THREADS
#include <pthread.h>
#include <gc.h>
#define UsedByGC 4
#define GC_CONFIG() GC_all_interior_pointers = 1

#define PtrCast(Num) ((void*) (Num))
#define IntCast(Ptr) ((intptr_t) (Ptr))

#define ClearFlag(Ptr, Flag) PtrCast(IntCast(Ptr) ^ GetFlag(Ptr, Flag))
#define GetFlag(Ptr, Flag) (IntCast(Ptr) & (Flag))
#define SetFlag(Ptr, Flag) ((Ptr) = PtrCast((IntCast(Ptr) | (Flag))))
#define TestFlag(Ptr, Flag) (GetFlag(Ptr, Flag) == (Flag))

#define Stringify(D) #D
#define StringifyValue(D) Stringify(D)

/* I should kill this darling. I don't even use it. */
#define block switch (0) default:

#endif /* CommonH */

```

Listing B.2: common.h

```

/* See license.txt for copyright and licensing details. */

#ifndef GlobalH
#define GlobalH

#define _Func(N) CreateGlobal ## N
#define _Name(N) Global ## N
#define _Ref(N) _Global ## N ## Reference

#define _Decl(N) _Name(N)* _Ref(N)
#define _Defn(T, N) typedef T _Name(N)
#define _Fwd(N) _Name(N)* _Func(N)(void)

```

```

#define _Get(N) (_Ref(N) != (void*)0) ? _Ref(N) : _Func(N)()

#ifdef Implementation
#define Global(T, N) _Defn(T, N); extern _Decl(N); _Fwd(N) {
    static _Name(N) Instance;
    _Ref(N) = &Instance;
    return _Ref(N);
}
_Decl(N) = (void*)0
#else
#define Global(T, N) _Defn(T, N); extern _Decl(N); _Fwd(N)
#endif

#define ModifiesGlobal(N) Global ## N* const N = _Get(N)
#define UsesGlobal(N) const Global ## N* const N = _Get(N)

#endif /* GlobalH */

```

Listing B.3: global.h

```

/* See license.txt for copyright and licensing details. */

#ifndef SettingsH
#define SettingsH

#include "global.h"

struct settings {
    char** ArgumentList;

    char* LineEditingStyle;
    char* Prompt;

    int ArgumentCount;
    int HistorySize;
    int IsScript;
};

Global(struct settings, Settings);

#endif /* SettingsH */

```

Listing B.4: settings.h

```

/* See license.txt for copyright and licensing details. */

```

```
#define Implementation
#include "settings.h"
```

Listing B.5: settings.c

```
/* See license.txt for copyright and licensing details. */

#ifndef LoggingH
#define LoggingH

#ifndef NDEBUG
#define Critical if (0) (void)
#define Error if (0) (void)
#define Warning if (0) (void)
#define Info if (0) (void)
#define Debug if (0) (void)
#else
#define Critical SetLogInfo(__FILE__, __LINE__, llCritical), LogMessage
#define Error SetLogInfo(__FILE__, __LINE__, llError), LogMessage
#define Warning SetLogInfo(__FILE__, __LINE__, llWarning), LogMessage
#define Info SetLogInfo(__FILE__, __LINE__, llInfo), LogMessage
#define Debug SetLogInfo(__FILE__, __LINE__, llDebug), LogMessage
#endif /* NDEBUG */

enum { llNone = 0, llCritical, llError, llWarning, llInfo, llDebug };

void LogMessage(char*, ...);
void SetLogFile(void*);
void SetLogInfo(char*, int, int);
void SetLogLevel(int);

#endif /* LoggingH */
```

Listing B.6: logging.h

```
/* See license.txt for copyright and licensing details. */

#include <stdio.h>
#include <stdarg.h>

#include "logging.h"

static int Level = llDebug;
static FILE* Log = NULL; /* NULL as stderr is not guaranteed to be constant. */
```

```

static char* Types[] = {"", "Critical", "Error", "Warning", "Info", "Debug"};

static char* Name;
static int Position = 0;
static int Severity = 0;

void LogMessage(char* Format, ...) {
    va_list Args;

    if (Severity > Level)
        return;

    fprintf((Log != NULL) ? Log : stderr,
            "%s|%s:%d|", Types[Severity], Name, Position);

    va_start(Args, Format);
    vfprintf((Log != NULL) ? Log : stderr, Format, Args);
    va_end(Args);

    return;
}

void SetLogFile(void* File) {
    Log = File;

    return;
}

void SetLogInfo(char* File, int Line, int Type) {
    Name = File;
    Position = Line;
    Severity = Type;

    return;
}

void SetLogLevel(int Type) {
    Level = Type;

    return;
}

```

Listing B.7: logging.c

B.3 Storage

```

CellType(Integer, int, "integer")
CellType(Status, int, "status")
CellType(Double, double, "double")
CellType(Symbol, char*, "symbol")
CellType(String, char*, "string")
CellType(Function, function, "function")
CellType(Pair, pair, "pair")
CellType(Method, pair, "method")
CellType(Env, pair, "env")
CellType(Scope, pair, "scope")
CellType(Channel, pair, "channel")
#undef CellType

```

/ See license.txt for copyright and licensing details. */*

Listing B.8: cell.def

/ See license.txt for copyright and licensing details. */*

```

#ifndef CellH
#define CellH

#include <stdio.h>

#define Car(Cell) ((pair*) (Cell))->Head
#define Cdr(Cell) ((pair*) (Cell))->Tail
#define Caar(Cell) Car(Car(Cell))
#define Cadr(Cell) Car(Cdr(Cell))
#define Cdar(Cell) Cdr(Car(Cell))
#define Cddr(Cell) Cdr(Cdr(Cell))
#define Caaar(Cell) Car(Caar(Cell))
#define Caadr(Cell) Car(Cadr(Cell))
#define Cadar(Cell) Car(Cdar(Cell))
#define Caddr(Cell) Car(Cddr(Cell))
#define Cdaar(Cell) Cdr(Caar(Cell))
#define Cdadr(Cell) Cdr(Cadr(Cell))
#define Cddar(Cell) Cdr(Cdar(Cell))
#define Cdddr(Cell) Cdr(Cddr(Cell))
#define Caaaar(Cell) Car(Caaar(Cell))
#define Caaadr(Cell) Car(Caadr(Cell))
#define Caadar(Cell) Car(Cadar(Cell))
#define Caaddr(Cell) Car(Caddr(Cell))
#define Cadaar(Cell) Car(Cdaar(Cell))

```



```

#define Cadadr(Cell) Car(Cdadr(Cell))
#define Caddar(Cell) Car(Cddar(Cell))
#define Caddr(Cell) Car(Cdddr(Cell))
#define Cdaaar(Cell) Cdr(Caaar(Cell))
#define Cdaadr(Cell) Cdr(Caadr(Cell))
#define Cdadar(Cell) Cdr(Cadar(Cell))
#define Cdaddr(Cell) Cdr(Caddr(Cell))
#define Cddaar(Cell) Cdr(Cdaar(Cell))
#define Cddadr(Cell) Cdr(Cdadr(Cell))
#define Cdddar(Cell) Cdr(Cddar(Cell))
#define Cdddr(Cell) Cdr(Cdddr(Cell))
#define Cons(Head, Tail) CreatePair(ctPair, Head, Tail, NULL)

/* Cell flags. */
enum { cfUserDefined = 1, cfTagged = 2 };

enum {
    ctNone = 0,
    ctBoolean = 1,
#define CellType(Name, Unused1, Unused2) ct ## Name,
#include "cell.def"
    ctMax };

typedef void (*finalizer)(void*, void*);
typedef int (*function)(void*, void*);

typedef struct pair {
    void* Head;
    void* Tail;
} pair;

extern pair* Nil;

extern void* False;
extern void* True;

/* CellInit must be called before any other functions are called. */
void CellInit(int Type, ...);
void CellExit(void);

void* CreateCell(int, finalizer);
pair* CreateList(void*, ...);
pair* CreatePair(int, void*, void*, finalizer);

/* Is<T> functions. */
#define CellType(Name, Unused1, Unused2) int Is ## Name(void*);

```

```

#include "cell.def"

#define IsText(C) (IsSymbol(C) || IsString(C))

int IsAtom(void*);
int IsBoolean(void*);
int IsCons(void*);
int IsList(void*);
int IsNull(void*);
int IsObject(void*);

/* Unwrap<T> functions. */
#define CellType(Name, Type, Unused) Type Unwrap ## Name(void*);
#include "cell.def"

short UnwrapBoolean(void*);

#define UnwrapText UnwrapString

/* Wrap<T> functions. */
#define CellType(Name, Type, Unused) Type* Wrap ## Name(Type);
#include "cell.def"

short* WrapBoolean(short);

#define WrapText WrapString

/* Cast functions. */
short ToBoolean(void*);
int ToInteger(void*);
int ToStatus(void*);
double ToDouble(void*);
char* ToString(void*);

pair* Append(void* CurrentList, ...);
pair* AppendTo(void* CurrentList, ...);
pair* Assoc(void*, void*);
int Equal(void*, void*);
int GetType(void* Cell);
pair* Join(void* CurrentList, ...);
pair* JoinTo(void* CurrentList, ...);
int Length(void*);
pair* ListRef(void*, int);
pair* ListTail(void*, int);
pair* Reverse(void*);

```

```

void Write(FILE*, void*);
void LowLevelWrite(int, void*);

#endif /* CellH */

```

Listing B.9: cell.h

```

/* See license.txt for copyright and licensing details. */

#include "common.h"

#include <assert.h>
#include <float.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "cell.h"
#include "logging.h"

typedef void* (*allocator)(size_t);
typedef int (*callback)(void*);

typedef struct boolean {
    short Tag;
    short Value;
} boolean;

typedef struct tagged {
    short Tag;
    char Value[];
} tagged;

static boolean _False = { ctBoolean, 0 };
static boolean _True = { ctBoolean, 1 };

static int BlockSize = 0;
static int PageMask = 0;
static int PageSize = 0;
static int DoNothing(void* Unused) { return 0; }

static struct {
    void* Data;
    char* Next;

```

```

    char* Maximum;
    int Size;
    int Type;
    callback Before;
    callback After;
    allocator Allocator;
} Block[] = {
    {NULL, NULL, NULL, 0, ctNone, DoNothing, DoNothing, NULL},
    {NULL, NULL, NULL, sizeof(short), ctBoolean, DoNothing, DoNothing, NULL},
#define CellType(Name, Type, Unused) \
    {NULL, NULL, NULL, sizeof(Type), ct ## Name, DoNothing, DoNothing, NULL},
#include "cell.def"
};

void* False = &_False;
void* True = &_True;

pair* Nil = NULL;

void CellExit(void) {
    return;
}

void CellInit(int Type, ...) {
    va_list Args;

    /* If the following isn't true, we're going to have some trouble. */
    assert(sizeof(int) == sizeof(void*));
    assert(offsetof(tagged, Value) == cfTagged);
    assert(offsetof(boolean, Value) == cfTagged);

    va_start(Args, Type);
    while (Type > ctNone && Type < ctMax) {
        Block[Type].Before = va_arg(Args, callback);
        Block[Type].After = va_arg(Args, callback);
        Block[Type].Data = va_arg(Args, void*);

        /* Get the the next type. */
        Type = va_arg(Args, int);
    }
    va_end(Args);

    /* Exit if we've already been here. */
    if (Nil != NULL)
        return;

```

```

/* PageSize must be power of 2. */
PageSize = getpagesize();
PageMask = PageSize - 1;
assert((PageMask & PageSize) == 0);

/* GC uses 4 bytes of page. */
BlockSize = PageSize - UsedByGC;

GC.CONFIG();

GC.INIT();

GC_enable_incremental();

/* Initialize allocators */
for (Type = ctNone; Type < ctMax; Type++) {
    assert(Type == Block[Type].Type);

    /* Blocks for Integers, Doubles, ... do not reference other memory. */
    Block[Type].Allocator = (Type <= ctDouble) ?
        GC_malloc_atomic : GC_malloc;
}

Nil = CreateCell(ctPair, NULL);
Nil->Head = Nil;
Nil->Tail = Nil;

return;
}

void* CreateCell(int Type, finalizer Finalizer) {
    char* FreshBlock = NULL;
    void* FreshCell = NULL;
    tagged* Tagged = NULL;

    assert(PageSize);
    assert(Type > ctBoolean);

    Block[Type].Before(Block[Type].Data);

    if (Type <= ctScope) {
        if (Block[Type].Next >= Block[Type].Maximum) {
            FreshBlock = Block[Type].Allocator(BlockSize);

            assert(FreshBlock != NULL);

```

```

        if (Finalizer)
            GC_register_finalizer(FreshBlock, Finalizer, NULL, NULL, NULL);

        *(int*) FreshBlock = Type;

        Block[Type].Next = FreshBlock + sizeof(int);
        Block[Type].Maximum =
            FreshBlock + BlockSize - Block[Type].Size + 1;
    }

    FreshCell = Block[Type].Next;
    Block[Type].Next += Block[Type].Size;
}
else {
    assert(Type < ctMax);

    Tagged = Block[Type].Allocator(Block[Type].Size + cfTagged);

    assert(Tagged != NULL);

    if (Finalizer)
        GC_register_finalizer(Tagged, Finalizer, NULL, NULL, NULL);

    Tagged->Tag = Type;

    FreshCell = &Tagged->Value;
}

Block[Type].After(Block[Type].Data);

return FreshCell;
}

/* Construct a list containing the given values. */
pair* CreateList(void* CurrentValue, ...) {
    pair* CurrentPair = NULL;
    pair* StartPair = Cons(CurrentValue, Nil);
    pair* PreviousPair = StartPair;

    va_list Args;
    va_start(Args, CurrentValue);
    while ((CurrentValue = va_arg(Args, void*)) {
        CurrentPair = Cons(CurrentValue, Nil);

        Cdr(PreviousPair) = CurrentPair;
        PreviousPair = CurrentPair;
    }
}

```

```

    }
    va_end(Args);

    return StartPair;
}

pair* CreatePair(int Type, void* Head, void* Tail, finalizer Finalizer) {
    pair* FreshPair = NULL;

    assert(Type >= ctPair);

    FreshPair = CreateCell(Type, Finalizer);
    FreshPair->Head = Head;
    FreshPair->Tail = Tail;

    return FreshPair;
}

/* Is<T> functions. */
#define CellType(Name, Unused1, Unused2) int Is ## Name(void* Cell) { \
    return ct ## Name == GetType(Cell); \
}
#include "cell.def"

int IsAtom(void* Cell) {
    return ctString >= GetType(Cell);
}

int IsBoolean(void* Cell) {
    return Cell == True || Cell == False;
}

int IsCons(void* Cell) {
    return ctPair <= GetType(Cell);
}

int IsList(void* Cell) {
    pair* Pair = Cell;

    while (IsCons(Pair) && !IsNull(Pair))
        Pair = Cdr(Pair);

    return IsNull(Pair);
}

int IsNull(void* Cell) {

```

```

    return Cell == (void*) Nil;
}

int IsObject(void* Cell) {
    return ctScope <= GetType(Cell);
}

/* Unwrap<T> functions. */
#define CellType(Name, Type, Unused) Type Unwrap ## Name(void* Cell) { \
    assert(Cell != NULL); \
    return *(Type*) Cell; \
}
#include "cell.def"

short UnwrapBoolean(void* Cell) {
    return Cell == True;
}

/* Wrap<T> functions. */
#define CellType(Name, Type, Unused) Type* Wrap ## Name(Type Value) { \
    Type* Fresh ## Name = CreateCell(ct ## Name, NULL); \
    *((Type*) Fresh ## Name) = Value; \
    return Fresh ## Name; \
}
#include "cell.def"

short* WrapBoolean(short Value) {
    return Value ? True : False;
}

short ToBoolean(void* Cell) {
    char* String = NULL;

    switch (GetType(Cell)) {
    case ctBoolean:
        return UnwrapBoolean(Cell);
    case ctInteger:
        return (UnwrapInteger(Cell) != 0);
    case ctStatus:
        return (UnwrapStatus(Cell) == 0);
    case ctDouble:
        return (UnwrapDouble(Cell) != 0.0);
    case ctSymbol:
        String = UnwrapSymbol(Cell);
        return (strcmp(String, "false") != 0);
    case ctString:

```



```

        return 1;
    default:
        return !IsNull(Cell);
    }
}

int ToInteger(void* Cell) {
    switch (GetType(Cell)) {
    case ctBoolean:
        return UnwrapBoolean(Cell);
    case ctInteger:
        return UnwrapInteger(Cell);
    case ctStatus:
        return UnwrapStatus(Cell);
    case ctDouble:
        return ((int) UnwrapDouble(Cell));
    case ctSymbol:
    case ctString:
        return ((int) strtol(UnwrapText(Cell), NULL, 10));
    default:
        return Length(Cell);
    }
}

int ToStatus(void* Cell) {
    switch (GetType(Cell)) {
    case ctBoolean:
        return (UnwrapBoolean(Cell) == 0);
    case ctInteger:
        return UnwrapInteger(Cell);
    case ctStatus:
        return UnwrapStatus(Cell);
    case ctDouble:
        return ((int) UnwrapDouble(Cell));
    case ctSymbol:
    case ctString:
        return ((int) strtol(UnwrapText(Cell), NULL, 10));
    default:
        return Length(Cell);
    }
}

double ToDouble(void* Cell) {
    switch (GetType(Cell)) {
    case ctBoolean:
        return ((double) UnwrapBoolean(Cell));

```

```

    case ctInteger:
        return ((double) UnwrapInteger(Cell));
    case ctStatus:
        return ((double) UnwrapStatus(Cell));
    case ctDouble:
        return UnwrapDouble(Cell);
    case ctSymbol:
    case ctString:
        return strtod(UnwrapText(Cell), NULL);
    default:
        return ((double) Length(Cell));
    }
}

char* ToString(void* Cell) {
    char Buffer[DBL_MAX_10_EXP - DBL_MIN_10_EXP];
    int Length = 0;

    switch (GetType(Cell)) {
    case ctBoolean:
        return GC_strdup((UnwrapBoolean(Cell)) ? "true" : "false");
    case ctInteger:
        Length = snprintf(Buffer, sizeof(Buffer), "%d", UnwrapInteger(Cell));
        assert(Length > 0);

        return GC_strdup(Buffer);
    case ctStatus:
        Length = snprintf(Buffer, sizeof(Buffer), "%d", UnwrapStatus(Cell));
        assert(Length > 0);

        return GC_strdup(Buffer);
    case ctDouble:
        Length = snprintf(Buffer, sizeof(Buffer), "%g", UnwrapDouble(Cell));
        assert(Length > 0);

        return GC_strdup(Buffer);
    case ctSymbol:
    case ctString:
        return UnwrapText(Cell);
    default:
        Length = snprintf(Buffer, sizeof(Buffer), "Cell(%p)", Cell);
        assert(Length > 0);

        return GC_strdup(Buffer);
    }
}

```

```

/*
 * Create a new list that is a copy of the current list with any arguments
 * added at the end of that list.
 */
pair* Append(void* CurrentList, ...) {
    pair* CurrentPair = NULL;
    pair* PreviousPair = NULL;
    pair* StartPair = Nil;
    void* CurrentValue = NULL;

    va_list Args;
    va_start(Args, CurrentList);

    if (CurrentList != NULL && !IsNull(CurrentList)) {
        StartPair = Cons(Car(CurrentList), Nil);
        PreviousPair = StartPair;

        CurrentList = Cdr(CurrentList);
        while (!IsNull(CurrentList)) {
            CurrentPair = Cons(Car(CurrentList), Nil);

            Cdr(PreviousPair) = CurrentPair;
            PreviousPair = CurrentPair;

            CurrentList = Cdr(CurrentList);
        }
    } else {
        StartPair = Cons(va_arg(Args, void*), Nil);
        PreviousPair = StartPair;
    }

    while ((CurrentValue = va_arg(Args, void*))) {
        CurrentPair = Cons(CurrentValue, Nil);

        Cdr(PreviousPair) = CurrentPair;
        PreviousPair = CurrentPair;
    }
    va_end(Args);

    return StartPair;
}

/* Destructively append any arguments to the end of the current list. */
pair* AppendTo(void* CurrentList, ...) {
    pair* CurrentPair = NULL;

```

```

pair* PreviousPair = NULL;
pair* StartPair = Nil;
void* CurrentValue = NULL;

va_list Args;
va_start(Args, CurrentList);

if (CurrentList != NULL && !IsNull(CurrentList)) {
    StartPair = CurrentList;

    PreviousPair = StartPair;
    CurrentList = Cdr(CurrentList);

    while (!IsNull(CurrentList)) {
        PreviousPair = CurrentList;
        CurrentList = Cdr(CurrentList);
    }
} else {
    StartPair = Cons(va_arg(Args, void*), Nil);
    PreviousPair = StartPair;
}

while ((CurrentValue = va_arg(Args, void*))) {
    CurrentPair = Cons(CurrentValue, Nil);

    Cdr(PreviousPair) = CurrentPair;
    PreviousPair = CurrentPair;
}
va_end(Args);

return StartPair;
}

pair* Assoc(void* AList, void* Key) {
    while (!IsNull(AList)) {
        if (Equal(Caar(AList), Key))
            return Car(AList);
        AList = Cdr(AList);
    }

    return Nil;
}

int Equal(void* A, void* B) {
    int Type = 0;

```

```

    if (IsNull(A) && IsNull(B))
        return 1;

    if (IsCons(A) && IsCons(B))
        return Equal(Car(A), Car(B)) && Equal(Cdr(A), Cdr(B));

    Type = GetType(A);
    if (Type != GetType(B))
        return 0;

    switch (Type) {
    case ctBoolean:
        return UnwrapBoolean(A) == UnwrapBoolean(B);
    case ctInteger:
        return UnwrapInteger(A) == UnwrapInteger(B);
    case ctStatus:
        return UnwrapStatus(A) == UnwrapStatus(B);
    case ctDouble:
        return UnwrapDouble(A) == UnwrapDouble(B);
    case ctSymbol:
    case ctString:
        return (strcmp(*(char**) A, *(char**) B) == 0);
    default:
        return 0;
    }
}

int GetType(void* Cell) {
    intptr_t Address = 0;
    tagged* Tagged = NULL;

    assert(PageMask);
    assert(Cell != NULL);

    if (IsBoolean(Cell))
        return ctBoolean;

    Address = IntCast(ClearFlag(Cell, cfUserDefined | cfTagged));

    if (TestFlag(Cell, cfTagged)) {
        Tagged = (tagged*) Address;
        return (int) Tagged->Tag;
    }
    else
        return *((int*) PtrCast(Address -= (Address & PageMask)));
}

```

```

/* Combine lists into one large list. */
pair* Join(void* CurrentList, ...) {
    pair* StartPair = Cons(Car(CurrentList), Nil);
    pair* PreviousPair = StartPair;
    va_list Args;

    CurrentList = Cdr(CurrentList);

    va_start(Args, CurrentList);
    do {
        while (!IsNull(CurrentList)) {
            pair* CurrentPair = Cons(Car(CurrentList), Nil);
            Cdr(PreviousPair) = CurrentPair;
            PreviousPair = CurrentPair;

            CurrentList = Cdr(CurrentList);
        }
    } while ((CurrentList = va_arg(Args, pair*)));
    va_end(Args);

    return StartPair;
}

/* Destructively combine lists into one large list. */
pair* JoinTo(void* CurrentList, ...) {
    pair* PreviousPair = NULL;

    pair* StartPair = CurrentList;

    va_list Args;
    va_start(Args, CurrentList);
    do {
        if (PreviousPair != NULL)
            Cdr(PreviousPair) = CurrentList;

        while (!IsNull(CurrentList)) {
            PreviousPair = CurrentList;
            CurrentList = Cdr(CurrentList);
        }
    } while ((CurrentList = va_arg(Args, pair*)));
    va_end(Args);

    return StartPair;
}

```

```

int Length(void* Cell) {
    int Length = 0;

    while (IsCons(Cell) && !IsNull(Cell)) {
        Cell = Cdr(Cell);
        Length++;
    }

    return Length;
}

pair* ListRef(void* Cell, int Index) {
    return Car(ListTail(Cell, Index));
}

pair* ListTail(void* Cell, int Index) {
    while (Index-- > 0 && IsCons(Cell)) {
        Cell = Cdr(Cell);
    }

    return Cell;
}

pair* Reverse(void* Cell) {
    pair* Reversed = Nil;

    while (IsCons(Cell) && !IsNull(Cell)) {
        Reversed = Cons(Car(Cell), Reversed);
        Cell = Cdr(Cell);
    }

    return Reversed;
}

static void _Write(int Depth, FILE* File, void* Cell)
{
    int Count = 0;
    int Type = 0;

    if (Cell == NULL) {
        fprintf(File, "%%[null]");
        return;
    }

    Type = GetType(Cell);
    switch (Type) {

```

```

case ctBoolean:
    if (UnwrapBoolean(Cell))
        fprintf(File, "true");
    else
        fprintf(File, "false");

    break;

case ctDouble:
    fprintf(File, "%g", UnwrapDouble(Cell));
    break;

case ctInteger:
case ctStatus:
    fprintf(File, "%d", UnwrapInteger(Cell));
    break;

case ctPair:
    if (IsNull(Cell)) {
        if (!Depth)
            fprintf (File, "'{'");
        else
            return;
    } else if (GetType(Cdr(Cell)) == ctPair) {
        if (Depth)
            fprintf(File, "{");

        while (GetType(Cell) == ctPair && !IsNull(Cell)) {
            if (Count++)
                fprintf(File, " ");
            _Write(Depth + 1, File, Car(Cell));
            Cell = Cdr(Cell);
        }
        _Write(Depth + 1, File, Cell);

        if (Depth)
            fprintf(File, "}");
    } else {
        _Write(Depth + 1, File, Car(Cell));
        fprintf(File, "::");
        _Write(Depth + 1, File, Cdr(Cell));
    }
    break;

case ctSymbol:
case ctString:

```



```

        fprintf(File, "%s", UnwrapText(Cell));
        break;

    case ctMethod:
        fprintf(File, "%%method %p%%", Cell);
        break;

    case ctEnv:
        fprintf(File, "%%env %p%%", Cell);
        break;

    case ctScope:
        fprintf(File, "%%scope %p%%", Cell);
        break;

    case ctChannel:
        fprintf(File, "%%channel %p%%", Cell);
        break;

    default:
        fprintf(File, "%%[unknown %d]", Type);
        break;
}

if (!Depth)
    fprintf(File, "\n");
}

void Write(FILE* File, void* Cell)
{
    _Write(0, File, Cell);
}

static void _LowLevelWrite(int Depth, char* Buffer, int* Size, void* Cell)
{
    int Count = 0;
    int Type = 0;

    if (Cell == NULL) {
        *Size += snprintf(Buffer + *Size, 8192, "%%[null]");
        return;
    }

    Type = GetType(Cell);
    switch (Type) {
    case ctBoolean:

```

```

    if (UnwrapBoolean(Cell))
        *Size += snprintf(Buffer + *Size, 8192, "true");
    else
        *Size += snprintf(Buffer + *Size, 8192, "false");

    break;

case ctDouble:
    *Size += snprintf(Buffer + *Size, 8192, "%g", UnwrapDouble(Cell));
    break;

case ctInteger:
case ctStatus:
    *Size += snprintf(Buffer + *Size, 8192, "%d", UnwrapInteger(Cell));
    break;

case ctPair:
    if (IsNull(Cell)) {
        if (!Depth)
            *Size += snprintf(Buffer + *Size, 8192, "'{}");
        else
            return;
    } else if (GetType(Cdr(Cell)) == ctPair) {
        if (Depth)
            *Size += snprintf(Buffer + *Size, 8192, "{");

        while (GetType(Cell) == ctPair && !IsNull(Cell)) {
            if (Count++)
                *Size += snprintf(Buffer + *Size, 8192, " ");
            _LowLevelWrite(Depth + 1, Buffer, Size, Car(Cell));
            Cell = Cdr(Cell);
        }
        _LowLevelWrite(Depth + 1, Buffer, Size, Cell);

        if (Depth)
            *Size += snprintf(Buffer + *Size, 8192, "}");
    } else {
        _LowLevelWrite(Depth + 1, Buffer, Size, Car(Cell));
        *Size += snprintf(Buffer + *Size, 8192, " :: ");
        _LowLevelWrite(Depth + 1, Buffer, Size, Cdr(Cell));
    }
    break;

case ctSymbol:
case ctString:
    *Size += snprintf(Buffer + *Size, 8192, "%s", UnwrapText(Cell));

```

```

        break;

    case ctMethod:
        *Size += snprintf(Buffer + *Size, 8192, "%%method %p%%", Cell);
        break;

    case ctEnv:
        *Size += snprintf(Buffer + *Size, 8192, "%%env %p%%", Cell);
        break;

    case ctScope:
        *Size += snprintf(Buffer + *Size, 8192, "%%scope %p%%", Cell);
        break;

    case ctChannel:
        *Size += snprintf(Buffer + *Size, 8192, "%%channel %p%%", Cell);
        break;

    default:
        *Size += snprintf(Buffer + *Size, 8192, "%%[unknown %d]", Type);
        break;
}

if (!Depth)
    *Size += snprintf(Buffer + *Size, 8192, "\n");
}

void LowLevelWrite(int File, void* Cell)
{
    char Buffer[8192];
    int Size = 0;
    int Status = 0;

    _LowLevelWrite(0, Buffer, &Size, Cell);

    Status = write(File, Buffer, Size);

    Debug("write returned %d\n", Status);
}

```

Listing B.10: cell.c

B.4 Parser

```

/* See license.txt for copyright and licensing details. */

#ifndef ParserH
#define ParserH

typedef struct parser parser;

parser* CreateParser(void*, int (*)(void*, void*));
int Parse(void*, int, void*);
void ProcessPhrase(parser*, void*);
void SetProcessPhraseData(parser*, void*);

#endif /*ParserH */

```

Listing B.11: parser.h

```

/* See license.txt for copyright and licensing details. */

#include "common.h"

#include <stdlib.h>

#include "parser.h"

struct parser {
    void* LemonHandle;
    void* ProcessPhraseData;
    int (*ProcessPhraseHook)(void*, void*);
};

void *LemonAlloc(void* (*)(size_t));
void Lemon(void*, int, void*, parser*);

static pthread_mutex_t ParserMutex = PTHREAD_MUTEX_INITIALIZER;

parser* CreateParser(void* ProcessPhraseData,
                    int (*ProcessPhraseHook)(void*, void*)) {
    parser* Parser = NULL;

    pthread_mutex_lock(&ParserMutex);

    Parser = GC_malloc(sizeof(*Parser));

```

```

    Parser->ProcessPhraseData = ProcessPhraseData;
    Parser->ProcessPhraseHook = ProcessPhraseHook;

    Parser->LemonHandle = LemonAlloc(&GC_malloc);

    pthread_mutex_unlock(&ParserMutex);

    return Parser;
}

int Parse(void* Parser, int Token, void* Value) {
    parser* P = (parser*) Parser;

    Lemon(P->LemonHandle, Token, Value, P);

    return 0;
}

void ProcessPhrase(parser* Parser, void* Phrase) {
    Parser->ProcessPhraseHook(Parser->ProcessPhraseData, Phrase);
}

void SetProcessPhraseData(parser* Parser, void* Data) {
    Parser->ProcessPhraseData = Data;
}

```

Listing B.12: parser.c

```

/* See license.txt for copyright and licensing details. */

#include {

#include "common.h"

#include <assert.h>
#include <stdlib.h>
#include <string.h>

#include "cell.h"
#include "parser.h"

char* OperatorName(char* Prefix, char* Operator) {
    int PrefixLength = strlen(Prefix);
    int OperatorLength = strlen(Operator) + 1;

    /* Assume this works. */

```

```

    char* Name = GC_malloc_atomic(PrefixLength + OperatorLength);

    memcpy(Name, Prefix, PrefixLength);
    memcpy(Name + PrefixLength, Operator, OperatorLength);

    return Name;
}

}

%syntax_error {
    printf("Syntax error!\n");
}

%extra_argument {parser* Parser}
%name Lemon
%token_type {void*}

/*
 * Names of operators: <Produces>_<Takes>_<Precedence>. Where <Takes> lists
 * the types ('C'ommand/'E'xpression) and the placement of the operator.
 *
 * Larger numbers indicate higher precedence.
 */

%left C_CO. /* Run in background: & */
%left C_COE. /* Redirection: <, >, >> !> !>> */
%left C_COC. /* Pipe: | !| */
%left E_EOE_0.
%left EQUALS_SIGN.
%right NOT.
%left E_EOE_1.
%left E_EOE_2. /* -or */
%left E_EOE_3.
%left E_EOE_4. /* -and */
%left E_EOE_5.
%left E_EOE_6. /* -eq, -is, -ne */
%left E_EOE_7.
%left E_EOE_8. /* -ge, -gt, -le, lt */
%left E_EOE_9.
%left E_EOE_A. /* +, - */
%left E_EOE_B.
%left E_EOE_C. /* -mul, -div, -mod */
%left E_EOE_D.
%right AT_SIGN.
%right SINGLE_QUOTE.

```

```

%right BACKQUOTE.
%left E_EOE_E.
%left DOUBLECOLON.
%left E_EOE_F.

program ::= top_block.

top_block ::= opt_evaluate_command.

top_block ::= top_block separator evaluate_command.

opt_evaluate_command ::= .

opt_evaluate_command ::= evaluate_command.

evaluate_command ::= command(C). {
    ProcessPhrase(Parser, C);
}

command(P) ::= command(C) C.CO(O). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), C, 0);
}

command(P) ::= command(L) C.COE(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), R, L, 0);
}

command(P) ::= command(L) C.COC(O) opt_separator command(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

command(P) ::= statement(S). { P = S; }

/* Multi-line statements end with the 'end' keyword. */
command(P) ::= statement(S) END. { P = S; }

/* After the 'end' keyword we can continue the list/statement. */
command(P) ::= statement(A) END statement(B). {
    P = JoinTo(A, B, 0);
}

statement(P) ::= list(L). { P = L; }

/* A substatement (whether on the same line or multi-line) must be last. */
statement(P) ::= list(L) sub_statement(S). {
    P = JoinTo(L, S, 0);
}

```

```

}

statement(P) ::= sub_statement(S). { P = S; }

/* Assignment is the only statement that operate on expressions. */
statement(P) ::= expression(L) EQUALS_SIGN opt_separator rhs(R). {
    P = JoinTo(CreateList(WrapSymbol("set"), L, 0), R, 0);
}

rhs(P) ::= expression(E). { P = CreateList(E, 0); }

/* This allows us to use '=' to set a variable to a statements value. */
rhs(P) ::= sub_statement(S). { P = S; }

/* This is a condensed substatement (like 'else: if ...') */
sub_statement(P) ::= COLON statement(S). { P = CreateList(S, 0); }

/* We can chain together multi-line statements. */
sub_statement(P) ::= COLON sub_block(B) statement(S). {
    P = JoinTo(B, S, 0);
}

sub_statement(P) ::= COLON sub_block(B). { P = B; }

sub_block(P) ::= separator INDENT block(B) separator DEDENT. { P = B; }

block(P) ::= command(C). { P = CreateList(C, 0); }

block(P) ::= block(B) separator command(C). {
    P = AppendTo(B, C, 0);
}

list(P) ::= expression(E). { P = Cons(E, Nil); }

list(P) ::= list(L) expression(E). {
    P = AppendTo(L, E, 0);
}

/* Nil is represented by '{}' - the empty set. */
expression(P) ::= LEFT_BRACE RIGHT_BRACE. { P = Nil; }

/* Braces are also how we convert a command into an expression. */
expression(P) ::= LEFT_BRACE command(C) RIGHT_BRACE. { P = C; }

expression(P) ::= expression(L) EEOE_0(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

```



```

}

expression(P) ::= expression(L) E_EOE_1(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_2(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_3(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_4(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_5(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_6(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_7(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_8(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_9(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_A(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) E_EOE_B(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

```

```

expression(P) ::= expression(L) EEOEC(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) EEOED(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) EEOEE(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= expression(L) EEOEF(O) opt_separator expression(R). {
    P = CreateList(WrapSymbol(OperatorName("op", O)), L, R, 0);
}

expression(P) ::= LEFT_PAREN expression(E) RIGHT_PAREN. { P = E; }

expression(P) ::= AT_SIGN expression(E). {
    P = CreateList(WrapSymbol("splice"), E, 0);
}

expression(P) ::= SINGLE_QUOTE expression(E). {
    P = CreateList(WrapSymbol("quote"), E, 0);
}

expression(P) ::= BACKQUOTE expression(E). {
    P = CreateList(WrapSymbol("backtick"), E, 0);
}

expression(P) ::= NOT expression(E). {
    P = CreateList(WrapSymbol("op!"), E, 0);
}

/* The double-colon, '::', is the 'cons' and object access operator. */
expression(P) ::= expression(L) DOUBLECOLON expression(R). {
    P = Cons(L, R);
}

expression(P) ::= PERCENT_SIGN BARE BARE(A) PERCENT_SIGN. {
    P = (void*) strtol((char*) A, NULL, 16);
}

expression(P) ::= word(W). { P = W; }

word(P) ::= QUOTED(W). {

```

```
    P = WrapString(W);
}

word(P) ::= BARE(B). {
    P = WrapSymbol(B);
}

opt_separator ::= .

opt_separator ::= separator.

separator ::= SEPARATOR.

separator ::= separator SEPARATOR.
```

Listing B.13: grammar.lemon

B.5 Scanner

```

/* See license.txt for copyright and licensing details. */

#ifndef ScannerH
#define ScannerH

typedef struct scanner scanner;

scanner* CreateScanner(void*, int (*)(void*, int, void*));
int Scan(void*, char *);

#endif /* ScannerH */

```

Listing B.14: scanner.h

```

/* See license.txt for copyright and licensing details. -*- mode: C -*- */

#include "common.h"

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "grammar.h"
#include "scanner.h"

#define YYCTYPE char
#define YYCURSOR Cursor
#define YYGETSTATE() Scanner->State
#define YYMARKER Scanner->Marker
#define YYSETSTATE(S) { Scanner->State = (S); }

#define yych Scanner->Current
#define yyaccept Scanner->Accept

#define FoundText(Scanner, Type, Cursor) { \
    CheckIndent(Scanner, Cursor); \
    SendToken(Scanner, Type, Cursor); \
    Reset(Scanner, Cursor); \
}

#define FoundToken(Scanner, Type, Cursor) { \
    SendToken(Scanner, Type, Cursor); \
}

```

```

        Reset(Scanner, Cursor);
    }

#define Reset(Scanner, Cursor) {
    Set(Scanner, Cursor);
    Scanner->WordSize = 0;
    goto yy0;
}

#define Set(Scanner, Cursor) { Scanner->Start = Cursor; }

struct scanner {
    int (*ProcessToken)(void*, int, void*);
    void* ProcessTokenData;

    YYCTYPE* Marker;
    YYCTYPE* Start;
    YYCTYPE* Word;

    int Accept;
    int CheckIndent;
    int Indent;
    int State;
    int WordMax;
    int WordSize;

    YYCTYPE Current;
};

static char* OperatorStrings[] = { "+", "-", "-and", "-div", "-eq",
    "-ge", "-gt", "-is", "-le", "-lt",
    "-mod", "-mul", "-ne", "-or", "end" };

static int Operators[] = { EEOE_A, EEOE_A, EEOE_4, EEOE_C, EEOE_6,
    EEOE_8, EEOE_8, EEOE_6, EEOE_8, EEOE_8,
    EEOE_C, EEOE_C, EEOE_6, EEOE_2, END };

static pthread_mutex_t ScannerMutex = PTHREAD_MUTEX_INITIALIZER;

static int CheckIndent(scanner* Scanner, char* Cursor) {
    int Indent = Scanner->Indent;

    if (!Scanner->CheckIndent)
        return 0;

    Scanner->Indent = 0;

```

```

while (Scanner->Start <= Cursor) {
    if (*Scanner->Start == '\t')
        Scanner->Indent += 4;
    else if (*Scanner->Start == ' ')
        Scanner->Indent++;
    else
        break;

    Scanner->Start++;
}

Indent = Scanner->Indent - Indent;
if (Indent > 0)
    Scanner->ProcessToken(Scanner->ProcessTokenData, INDENT, NULL);
else if (Indent < 0)
    Scanner->ProcessToken(Scanner->ProcessTokenData, DEDENT, NULL);

Scanner->CheckIndent = 0;

return Indent;
}

static int Compare(const void* A, const void* B) {
    return strcmp(*(char**) A, *(char**) B);
}

static int CheckWord(int Type, char* Word) {
    char** Found = NULL;

    if (Type != BARE)
        return Type;

    Found = bsearch(&Word, OperatorStrings, 15, sizeof(char*), Compare);
    if (Found == NULL)
        return Type;

    return Operators[Found - OperatorStrings];
}

static void SendToken(scanner* Scanner, int Type, char* Cursor) {
    char* Word = NULL;

    Scanner->WordSize = Cursor - Scanner->Start;

    if (Scanner->WordSize > Scanner->WordMax) {
        Scanner->WordMax = Scanner->WordSize;
    }
}

```

```

        Scanner->Word = GC_realloc(Scanner->Word, Scanner->WordMax + 1);
    }

    memcpy(Scanner->Word, Scanner->Start, Scanner->WordSize);

    Word = Scanner->Word;
    if (Type == QUOTED) {
        Scanner->WordSize--;
        Word++;
    }
    else if (Type == SEPARATOR)
        Scanner->CheckIndent = 1;

    Scanner->Word[Scanner->WordSize] = '\0';

    Type = CheckWord(Type, Word);

    Scanner->ProcessToken(Scanner->ProcessTokenData, Type, GC_strdup(Word));

    return;
}

scanner* CreateScanner(void* ProcessTokenData,
                      int (*ProcessToken)(void*, int, void*)) {
    scanner* Scanner = NULL;

    pthread_mutex_lock(&ScannerMutex);

    Scanner = GC_malloc(sizeof(*Scanner));

    memset(Scanner, 0, sizeof(*Scanner));

    Scanner->ProcessTokenData = ProcessTokenData;
    Scanner->ProcessToken = ProcessToken;

    Scanner->WordMax = 256;
    Scanner->Word = GC_malloc(Scanner->WordMax);

    pthread_mutex_unlock(&ScannerMutex);

    return Scanner;
}

int Scan(void* S, char* Cursor) {
    scanner* Scanner = (scanner*) S;

```

```

Scanner->Marker = Cursor;
Set(Scanner, Cursor);

for(;;) {
    /*!re2c
        re2c:yyfill:enable = 0;

        ESC = [\|] .;
        L = ESC | [$*+,-./0-9?A-Z[\|^_a-z~];
        W = [ \t];

        (["] (ESC | [^\"])* ["]) { FoundText(Scanner, QUOTED, Cursor); }
        L+ { FoundText(Scanner, BARE, Cursor); }
        W+ { CheckIndent(Scanner, Cursor); Reset(Scanner, Cursor); }
        W* "#" [^\n]* "\n" { FoundToken(Scanner, SEPARATOR, Cursor); }

        "\n" { FoundToken(Scanner, SEPARATOR, Cursor); }
        "!" { FoundToken(Scanner, NOT, Cursor); }
        "!>>" { FoundToken(Scanner, C_COE, Cursor); }
        "!>" { FoundToken(Scanner, C_COE, Cursor); }
        "!" { FoundToken(Scanner, C_COC, Cursor); }
        "%" { FoundToken(Scanner, PERCENT_SIGN, Cursor); }
        "&" { FoundToken(Scanner, C_CO, Cursor); }
        "(" { FoundToken(Scanner, LEFT_PAREN, Cursor); }
        ")" { FoundToken(Scanner, RIGHT_PAREN, Cursor); }

        "::" { FoundToken(Scanner, DOUBLE_COLON, Cursor); }
        ":" { FoundToken(Scanner, COLON, Cursor); }
        ";" { FoundToken(Scanner, SEPARATOR, Cursor); }
        "<" { FoundToken(Scanner, C_COE, Cursor); }
        "=" { FoundToken(Scanner, EQUALS_SIGN, Cursor); }
        ">" { FoundToken(Scanner, C_COE, Cursor); }
        ">>" { FoundToken(Scanner, C_COE, Cursor); }
        "@" { FoundToken(Scanner, AT_SIGN, Cursor); }
        "'" { FoundToken(Scanner, SINGLE_QUOTE, Cursor); }
        "\"" { FoundToken(Scanner, BACKQUOTE, Cursor); }
        "{" { FoundToken(Scanner, LEFT_BRACE, Cursor); }
        "|" { FoundToken(Scanner, C_COC, Cursor); }
        "}" { FoundToken(Scanner, RIGHT_BRACE, Cursor); }

        "\000" { return 0; }
        [^] { Reset(Scanner, Cursor); }
    */
}

```



```
    return 0;  
}
```

Listing B.15: scanner.c

B.6 Engine

```

/* See license.txt for copyright and licensing details. */

#ifndef EnvH
#define EnvH

#include "cell.h"

void* ApplyEnv(void*, void*, int);
void* CreateEnv(void*);
void ExtendEnv(void*, void*, void*);

void* AccessMember(void*, void*, int);
void AddMember(void*, void*, void*, int);
void* CopyObject(void*);
void* CreateObject(int, void*, finalizer);

#endif /* EnvH */

```

Listing B.16: env.h

```

/* See license.txt for copyright and licensing details. */

#include "common.h"

#include <assert.h>
#include <stdlib.h>
#include <unistd.h>

#include "cell.h"
#include "env.h"
#include "logging.h"

void* ApplyEnv(void* Env, void* Key, int CurrentOnly) {
    void* Value = Nil;

    assert(IsEnv(Env));

    for (; Env != Nil; Env = CurrentOnly ? Nil : Cdr(Env)) {
        Value = Assoc(Car(Env), Key);
        if (Value != Nil)
            break;
    }
}

```

```

    return Value;
}

static void* CopyEnv(void* Env) {
    void* NewEnv = Nil;
    void* Current = Nil;

    assert(IsEnv(Env));

    if (!IsNull(Car(Env))) {
        NewEnv = Append(Car(Env), 0);
        Current = NewEnv;
        while (!IsNull(Current)) {
            Car(Current) = Cons(Caar(Current), Cdar(Current));
            Current = Cdr(Current);
        }
    }

    return CreatePair(ctEnv, NewEnv,
                     IsNull(Cdr(Env)) ? Nil : CopyEnv(Cdr(Env)), NULL);
}

void* CreateEnv(void* PreviousEnv) {
    if (PreviousEnv == NULL)
        PreviousEnv = Nil;

    return CreatePair(ctEnv, Nil, PreviousEnv, NULL);
}

void ExtendEnv(void* Env, void* Key, void* Value) {
    void* Entry = ApplyEnv(Env, Key, 1);

    assert(IsEnv(Env));

    if (Entry == Nil)
        Car(Env) = CreatePair(ctEnv, Cons(Key, Value), Car(Env), NULL);
    else
        Cdr(Entry) = Value;

    return;
}

void* AccessMember(void* Object, void* Key, int PublicOnly) {
    void* Value = Nil;

    assert(IsObject(Object));

```

```

    for (; Object != Nil; Object = Cdr(Object)) {
        Value = ApplyEnv(PublicOnly ? Cdar(Object) : Car(Object), Key, 0);
        if (Value != Nil)
            break;
    }

    return Value;
}

void AddMember(void* Object, void* Key, void* Value, int Public) {
    assert(IsObject(Object));

    ExtendEnv(Public ? Cdar(Object) : Car(Object), Key, Value);

    return;
}

void* CopyObject(void* Object) {
    assert(IsObject(Object));

    return CreatePair(ctScope, CopyEnv(Car(Object)), Cdr(Object), NULL);
}

void* CreateObject(int Type, void* Context, finalizer Finalizer) {
    if (Context == NULL)
        Context = Nil;

    return CreatePair(Type, CreateEnv(CreateEnv(Nil)), Context, Finalizer);
}

```

Listing B.17: env.c

```

ProcessType(Code)
ProcessType(Env)
ProcessType(Scope)
ProcessType(Scratch)
ProcessType(Stack)
#undef ProcessType

/* See license.txt for copyright and licensing details. */

```

Listing B.18: process.def

```

/* See license.txt for copyright and licensing details. */

```

```

#ifndef EngineH
#define EngineH

typedef struct engine engine;

void EngineInit(void);
int ProcessCommand(void*, void*);

#endif /* EngineH */

```

Listing B.19: engine.h

```

/* See license.txt for copyright and licensing details. */

#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/wait.h>

#include <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <glob.h>
#include <limits.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "cell.h"
#include "common.h"
#include "engine.h"
#include "env.h"
#include "logging.h"
#include "parser.h"
#include "scanner.h"
#include "settings.h"

#define BindConsCellAccess(Type) \
    BindFunction(TopLevelScope, C ## Type ## rCommand, "c" #Type "r", 0)

#define BindContinuation(Process) \
    AddMember(Process->Scope, WrapSymbol("return"), \
              CreatePair(ctMethod, \
                          WrapInteger(psEvalReturnCommand), \
                          CreateList( Nil, Nil, \

```

```

                                CreateList(Cdr(Process->Scratch), \
                                                Process->Stack, 0), \
                                0), NULL), \
1)

#define BindFunction(Scope, Function, Name, Public) \
    AddMember(Scope, WrapSymbol(Name), \
               CreatePair(ctMethod, WrapFunction(Function), \
                           CreateList(Scope, Scope, 0), NULL), \
               Public)

#define BindState(Scope, State, Name, Public) \
    AddMember(Scope, WrapSymbol(Name), \
               CreatePair(ctMethod, WrapInteger(State), \
                           CreateList(Scope, Scope, 0), NULL), \
               Public)

#define CreateMethod(Scope, Method, Parameters) \
    CreatePair(ctMethod, Method, CreateList(Scope, Scope, Parameters, 0), NULL)

#define CreateScope(Previous) CreateObject(ctScope, Previous, NULL)

#define ExtendScope(Scope, Key, Value, Public) \
    AddMember(ClearFlag(Scope, stPublic), Key, Value, Public)

#define GetMethodBody(Method) Car(Method)
#define GetMethodParameters(Method) Caddr(Method)
#define GetMethodSelf(Method) Caddr(Method)
#define GetMethodScope(Method) Cadr(Method)

/* Process states. The save (sv*) states can be combined to form a state. */
enum {
    psNone = 0,

#define ProcessType(Name) sv ## Name = 1 << (--LINE-- - 1),
#include "process.def"

    svMax,

    psEvalAccess, psEvalAccess1,
    psEvalApplication, psEvalApplication1,
    psEvalArguments, psEvalArguments1,
    psEvalBlock, psEvalBlock1,
    psEvalCommand, psEvalCommand1,
    psEvalElement,
    psEvalReference, psEvalReference1, psEvalReference2,

```

```

    psEvalBuiltIn , psEvalExternal ,

    psEvalBlockCommand ,
    psEvalDefineCommand , psEvalDefineCommand1 ,
    psEvalDynamicCommand , psEvalDynamicCommand1 ,
    psEvalForCommand , psEvalForCommand1 , psEvalForCommand2 ,
    psEvalIfCommand , psEvalIfCommand1 ,
    psEvalImportCommand ,
    psEvalMethodCommand ,
    psEvalObjectCommand , psEvalObjectCommand1 ,
    psEvalPublicCommand , psEvalPublicCommand1 ,
    psEvalQuoteCommand ,
    psEvalReturnCommand ,
    psEvalSetCommand , psEvalSetCommand1 ,
    psEvalSpawnCommand ,
    psEvalSpliceCommand , psEvalSpliceCommand1 ,
    psEvalWhileCommand , psEvalWhileCommand1 , psEvalWhileCommand2 ,

    psEvalBacktick ,
    psEvalPipe , psEvalPipeParent , psEvalPipeChild ,
    psEvalPipeStderr , psEvalPipeStderrChild ,
    psEvalRedirectStdin , psEvalRedirectStdin1 , psEvalRedirectStdin2 ,
    psEvalRedirectStdout , psEvalRedirectStdout1 , psEvalRedirectStdout2 ,
    psEvalRedirectStderr , psEvalRedirectStderr1 , psEvalRedirectStderr2 ,
    psEvalRedirectAppendStdout , psEvalRedirectAppendStdout1 ,
    psEvalRedirectAppendStderr , psEvalRedirectAppendStderr1 ,

    psMax
};

/* Scope flag. */
enum { stPublic = cfUserDefined };

typedef struct {
    int IsFile;
    int Pipe[2];
    FILE* ReadEnd;
    FILE* WriteEnd;
    parser* Parser;
    scanner* Scanner;
    pthread_cond_t Full;
    pthread_cond_t Empty;
    pthread_mutex_t Mutex;
    pthread_mutex_t ReaderMutex;
    pthread_mutex_t WriterMutex;

```

```

} channel;

typedef struct {
#define ProcessType(Name) void* Name;
#include "process.def"
} process;

/* Global variables. */
static process* MainProcess = NULL;

#define CellType(Name, Unused1, Unused2) \
    static pthread_mutex_t Name ## Mutex = PTHREAD_MUTEX_INITIALIZER;
#include "cell.def"

static pthread_mutex_t CreateChannelMutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t CreateProcessMutex = PTHREAD_MUTEX_INITIALIZER;

/*****

    Forward Declarations.

*****/

static char* Chdir(process*, char*);
static channel* GetChannel(void*);
static int ImportFile(process*);
static int ReadFromChannel(void*, void*);
static void* Resolve(void*, void*, void*);
static int WriteToChannel(void*, void*);

/*****

    Helper Functions.

*****/

static void FinalizeChannel(void* ChannelObject, void* Unused) {
    channel* Channel = GetChannel(ChannelObject);

    printf("Finalizing channel\n");

    if (Channel->ReadEnd != NULL) {
        fclose(Channel->ReadEnd);
        Channel->ReadEnd = NULL;
    }
}

```



```

    }

    if (Channel->WriteEnd != NULL) {
        fclose(Channel->WriteEnd);
        Channel->WriteEnd = NULL;
    }

    return;
}

static void* GetArguments(process* P) {
    void* List = Nil;

    void* Element = Car(P->Scratch);
    while (Element != NULL) {
        List = Cons(Element, List);

        P->Scratch = Cdr(P->Scratch);
        Element = Car(P->Scratch);
    }

    if (Element == NULL)
        P->Scratch = Cdr(P->Scratch);

    return List;
}

static void* GetValue(void* Env, void* Scope, char* Name) {
    return Cdr(Resolve(Env, Scope, WrapSymbol(Name)));
}

static char* GetWorkingDir(void) {
    char* WorkingDir = GC_malloc_atomic(PATHMAX);

    return getcwd(WorkingDir, PATHMAX);
}

static void* Resolve(void* Env, void* Scope, void* Key) {
    void* Value = NULL;

    int Public = TestFlag(Scope, stPublic);

    Scope = ClearFlag(Scope, stPublic);

    Debug("Scope = %p, Public = %d\n", Scope, Public);

```

```

Value = AccessMember(Scope, Key, Public);
if (IsNull(Value)) {
    if (Env != NULL) {
        Value = ApplyEnv(Env, Key, 0);
        if (IsNull(Value))
            Value = Cons(Key, Key);
    }
}
else if (IsMethod(Cdr(Value))) {
    Debug("Setting method %s's self to %p\n", UnwrapText(Key), Scope);

    GetMethodSelf(Cdr(Value)) = Scope;
}

if (IsNull(Value)) {
    Error("Undefined member!\n");
    exit(0);
}

return Value;
}

static int SetCommand(void* Cell, void* Command) {
    void** C = (void**) Cell;

    *C = Command;

    return 1;
}

static int SetValue(void* Cell, void* Command) {
    void** C = (void**) Cell;

    *C = Car(Command);

    return 1;
}

/*****

    Channel Functions.

*****/

static channel* CreateChannel(int IsFile, int ReadEnd, int WriteEnd) {

```

```

channel* Channel = GC_malloc(sizeof(*Channel));

int Flags = 0;
int Status = 0;

pthread_cond_init(&Channel->Empty, NULL);
pthread_cond_init(&Channel->Full, NULL);

pthread_mutex_init(&Channel->Mutex, NULL);
pthread_mutex_init(&Channel->ReaderMutex, NULL);
pthread_mutex_init(&Channel->WriterMutex, NULL);

Channel->Parser = NULL;
Channel->Scanner = NULL;

Channel->IsFile = IsFile;
Channel->ReadEnd = NULL;
Channel->WriteEnd = NULL;

if (ReadEnd || WriteEnd) {
    Channel->Pipe[0] = ReadEnd;
    Channel->Pipe[1] = WriteEnd;
}
else {
    Status = pipe(Channel->Pipe);
    if (Status)
        Error("pipe failed (%d)\n", Status);
}

if (Channel->Pipe[0] >= 0) {
    Flags = fcntl(Channel->Pipe[0], F_GETFL, 0);
    Flags |= O_SYNC;

    fcntl(Channel->Pipe[0], F_SETFL, Flags);

    Channel->ReadEnd = fdopen(Channel->Pipe[0], "r");
    setlinebuf(Channel->ReadEnd);
}

if (Channel->Pipe[1] >= 0) {
    Flags = fcntl(Channel->Pipe[1], F_GETFL, 0);
    Flags |= O_SYNC;

    fcntl(Channel->Pipe[1], F_SETFL, Flags);

    Channel->WriteEnd = fdopen(Channel->Pipe[1], "w");
}

```

```

        setlinebuf(Channel->WriteEnd);
    }

    Debug("Read end of pipe (%p) is %d\n", Channel, Channel->Pipe[0]);
    Debug("Write end of pipe (%p) is %d\n", Channel, Channel->Pipe[1]);

    return Channel;
}

static void* CreateChannelObject(process* P, int IsFile,
                                int ReadEnd, int WriteEnd) {
    channel* Channel = NULL;
    void* ChannelObject = NULL;

    pthread_mutex_lock(&CreateChannelMutex);

    Channel = CreateChannel(IsFile, ReadEnd, WriteEnd);

    ChannelObject = CreateObject(ctChannel, P->Scope, FinalizeChannel);

    Debug("Channel object (%p)\n", ChannelObject);

    BindFunction(ChannelObject, ReadFromChannel, "read", 1);
    BindFunction(ChannelObject, WriteToChannel, "write", 1);
    ExtendScope(ChannelObject, WrapSymbol("guts"), Channel, 1);

    SetFlag(ChannelObject, stPublic);

    pthread_mutex_unlock(&CreateChannelMutex);

    return ChannelObject;
}

static channel* GetChannel(void* ChannelObject) {
    return Cdr(Resolve(nil, ChannelObject, WrapSymbol("guts")));
}

static int ReadFromChannel(void* Process, void* Unused) {
    char Buffer[1024];

    struct timespec Time;

    process* P = (process*) Process;
    void* Scope = GetMethodScope(Car(P->Scratch));
    void* Value = Nil;

```

```

channel* Channel = GetChannel(Scope);

int Max = 1000000;
int Size = 0;
int Status = 0;
int Wait = 262144;

if (Channel->ReadEnd == NULL || Channel->Pipe[0] == -1)
    return 1;

pthread_mutex_lock(&Channel->ReaderMutex);

if (Channel->Parser != NULL)
    SetProcessPhraseData(Channel->Parser, &Value);
else {
    Channel->Parser = CreateParser(&Value, SetValue);
    Channel->Scanner = CreateScanner(Channel->Parser, Parse);
}

if (Channel->Pipe[1] != -1) {
    pthread_mutex_lock(&Channel->Mutex);

    clock_gettime(CLOCK_REALTIME, &Time);
    ioctl(Channel->Pipe[0], FIONREAD, &Size);

    while (Size == 0) {
        Time.tv_nsec += Wait;
        if (Time.tv_nsec > 1000000000) {
            Time.tv_nsec -= 1000000000;
            Time.tv_sec += 1;
        }

        if (pthread_cond_timedwait(&Channel->Full, &Channel->Mutex,
                                   &Time) == ETIMEDOUT && Wait < Max)
            Wait *= 2;

        ioctl(Channel->Pipe[0], FIONREAD, &Size);
    }
}

Debug("Starting to read from channel\n");

while(IsNull(Value)) {
    Status = read(Channel->Pipe[0], Buffer, Size);

    Debug("read returned %d\n", Status);
}

```

```

        Buffer[Size] = '\0';

        Scan(Channel->Scanner, Buffer);
    }

    Debug("ReadFromChannel: process=%x channel=%x value=%s\n",
        Process, Channel, ToString(Value));

    Car(P->Scratch) = Value;

    ioctl(Channel->Pipe[0], FIONREAD, &Size);
    if (Size > 0)
        Info("More to be read\n");

    if (Channel->Pipe[1] != -1) {
        pthread_cond_signal(&Channel->Empty);
        pthread_mutex_unlock(&Channel->Mutex);
    }

    pthread_mutex_unlock(&Channel->ReaderMutex);

    return 1;
}

static int WriteToChannel(void* Process, void* Arguments) {
    struct timespec Time;

    process* P = (process*) Process;
    void* Scope = GetMethodScope(Car(P->Scratch));
    channel* Channel = GetChannel(Scope);

    int Max = 1000000;
    int Size = 0;
    int Wait = 262144;

    Debug("WriteToChannel: process=%x channel=%x value=%s\n",
        Process, Channel, ToString(Car(Arguments)));

    if (Channel->WriteEnd == NULL || Channel->Pipe[1] == -1)
        return 1;

    pthread_mutex_lock(&Channel->WriterMutex);

    Car(P->Scratch) = Arguments;

```

```

LowLevelWrite(Channel->Pipe[1], Arguments);

if (Channel->Pipe[0] != -1) {
    pthread_mutex_lock(&Channel->Mutex);

    pthread_cond_signal(&Channel->Full);

    clock_gettime(CLOCK_REALTIME, &Time);
    ioctl(Channel->Pipe[0], FIONREAD, &Size);

    Info("Time = %d.%d\n", Time.tv_sec, Time.tv_nsec);

    while (Size > 0) {
        Time.tv_nsec += Wait;
        if (Time.tv_nsec > 1000000000){
            Time.tv_nsec -= 1000000000;
            Time.tv_sec += 1;
        }

        Info("Waiting %d.%d\n", Time.tv_sec, Time.tv_nsec);
        if (pthread_cond_timedwait(&Channel->Empty, &Channel->Mutex,
                                   &Time) == ETIMEDOUT && Wait < Max)
            Wait *= 2;

        ioctl(Channel->Pipe[0], FIONREAD, &Size);
    }

    pthread_mutex_unlock(&Channel->Mutex);
}

pthread_mutex_unlock(&Channel->WriterMutex);

Info("Done\n", Wait);

return 1;
}

/*****

    State Functions.

*****/

static int GetState(process* Process) {
    return UnwrapInteger(Car(Process->Stack));
}

```

```

}

static void NewState(process* Process, int State) {
    Process->Stack = Cons(WrapInteger(State), Process->Stack);
}

static void RemoveState(process* Process) {
    Process->Stack = Cdr(Process->Stack);
}

static void ReplaceState(process* Process, int State) {
    Car(Process->Stack) = WrapInteger(State);
}

static void RestoreState(process* Process) {
    int RestoreFlags = UnwrapInteger(Car(Process->Stack));

    if (RestoreFlags & svScratch) {
        Process->Stack = Cdr(Process->Stack);
        Process->Scratch = Car(Process->Stack);
    }

    if (RestoreFlags & svScope) {
        Process->Stack = Cdr(Process->Stack);
        Process->Scope = Car(Process->Stack);
    }

    if (RestoreFlags & svEnv) {
        Process->Stack = Cdr(Process->Stack);
        Process->Env = Car(Process->Stack);
    }

    if (RestoreFlags & svCode) {
        Process->Stack = Cdr(Process->Stack);
        Process->Code = Car(Process->Stack);
    }

    if (RestoreFlags & svEnv)
        Chdir(Process, ".");

    return;
}

static int SaveState(process* Process, int SaveFlags, ...) {
    va_list Args;

```



```

    int CurrentState = GetState(Process);
    if (CurrentState == SaveFlags) {
        Debug("Not Saving State!\n");
        return 0;
    }

    va_start(Args, SaveFlags);

    if (SaveFlags & svCode)
        Process->Stack = Cons(va_arg(Args, void*), Process->Stack);

    if (SaveFlags & svEnv)
        Process->Stack = Cons(va_arg(Args, void*), Process->Stack);

    if (SaveFlags & svScope)
        Process->Stack = Cons(va_arg(Args, void*), Process->Stack);

    if (SaveFlags & svScratch)
        Process->Stack = Cons(va_arg(Args, void*), Process->Stack);

    va_end(Args);

    Process->Stack = Cons(WrapInteger(SaveFlags), Process->Stack);

    return 1;
}

/*****

    Evaluator.

*****/

static process* CreateProcess(void* InitialEnv, void* InitialScope) {
    process* Process = NULL;

    pthread_mutex_lock(&CreateProcessMutex);

    Process = GC_malloc(sizeof(*Process));

    memset(Process, 0, sizeof(*Process));

#define ProcessType(Name) Process->Name = Nil;
#include "process.def"

```

```

Process->Env = CreateEnv(InitialEnv);
Process->Scope = CreateScope(InitialScope);

Process->Stack = Nil;

pthread_mutex_unlock(&CreateProcessMutex);

return Process;
}

int External(void* Process, void* Arguments) {
    process* P = (process*) Process;

    pid_t ProcessID = 0;

    /*
     * We should check that we don't exceed the size of this
     * array and if we do we should increase (double) it's size
     */
    char* Args[1024];

    int ArgCount = 1;
    int Flags = GLOBNOCHECK | /* GLOB_QUOTE | */ GLOB_TILDE;
    int Status = 0;
    int Stdin = 0;
    int Stdout = 1;
    int Stderr = 2;

    Args[0] = UnwrapText(Car(P->Scratch));

    while (!IsNull(Arguments)) {
        if (IsString(Car(Arguments)))
            Args[ArgCount++] = ToString(Car(Arguments));
        else {
            glob_t Expanded;
            int Index = 0;

            glob(ToString(Car(Arguments)), Flags, NULL, &Expanded);
            Debug("Next argument matches: %d\n", Expanded.gl_pathc);
            for (Index = 0; Index < Expanded.gl_pathc; Index++)
                Args[ArgCount++] = GC_strdup(Expanded.gl_pathv[Index]);
            globfree(&Expanded);
        }
        Arguments = Cdr(Arguments);
    }
}

```

```

Args[ArgCount] = NULL;

Stdin = GetChannel(GetValue(P->Env, P->Scope, "$stdin"))->Pipe[0];
Stdout = GetChannel(GetValue(P->Env, P->Scope, "$stdout"))->Pipe[1];
Stderr = GetChannel(GetValue(P->Env, P->Scope, "$stderr"))->Pipe[1];

ProcessID = fork();

/* Error. */
if (ProcessID < 0) {
    Car(P->Scratch) = False;

    return 1;
}

/* Parent. */
if (ProcessID > 0) {
    waitpid(ProcessID, &Status, 0);

    Debug("Finished: %s (%d)\n", Args[0], WEXITSTATUS(Status));

    if (WIFEXITED(Status) && WEXITSTATUS(Status) == 127) {
        void* ChannelObject = GetValue(P->Env, P->Scope, "$stderr");
        void* WriteMethod = GetValue(nil, ChannelObject, "write");
        void* List = CreateList(WrapSymbol("choc: command not found:"),
                                WrapSymbol(Args[0]), 0);

        Car(P->Scratch) = WriteMethod;

        assert((UnwrapFunction(GetMethodBody(WriteMethod)))(P, List));
    }

    Car(P->Scratch) = WrapStatus(Status);

    return 1;
}

/* Child. */
Debug("%s: Stdin = %d, Stdout = %d, Stderr = %d\n",
      Args[0], Stdin, Stdout, Stderr);

Stdin = dup2(Stdin, STDIN_FILENO);
Stdout = dup2(Stdout, STDOUT_FILENO);

```

```

Stderr = dup2(Stderr, STDERR_FILENO);

Status = chdir(UnwrapText(GetValue(P->Env, P->Scope, "$cwd")));
if (Status)
    Warning("chdir failed (%d)\n", Status);

Debug("%s: Stdin = %d, Stdout = %d, Stderr = %d\n",
      Args[0], Stdin, Stdout, Stderr);

execvp(Args[0], Args);

/* This _should_ be a value we can distinguish from other exit codes. */
exit(127);

/* We should never reach here. */
return 1;
}

static int EvalBuiltIn(process* P) {
    void* Arguments = GetArguments(P);

    void* Method = Car(P->Scratch);
    void* Body = GetMethodBody(Method);

    return (UnwrapFunction(Body))(P, Arguments);
}

static void* ThreadFunction(void* Process) {
    char Line[1024];
    pthread_t ChildThread;

    char* Buffer;
    FILE* File = NULL;
    channel* Channel = NULL;
    process* ChildProcess = NULL;
    process* P = (process*) Process;

    void* Arguments = NULL;
    void* Body = NULL;
    void* ChannelObject = NULL;
    void* Method = NULL;
    void* Parameters = NULL;
    void* Self = NULL;

    int Index = 0;
    int State = 0;

```

```

while (!IsNull(P->Stack)) {
    State = GetState(P);

/*
    fprintf(stderr, "Stack: ");
    Write(stderr, P->Stack);
    fprintf(stderr, "Scratch: ");
    Write(stderr, P->Scratch);
    fprintf(stderr, "Code: ");
    Write(stderr, P->Code);
*/

switch (State) {
case psNone:
    RemoveState(P);
    return P->Scratch;

case psEvalBlock:
    P->Scratch = Cons(Nil, P->Scratch);

    ReplaceState(P, psEvalBlock1);

    /* Continue. */
case psEvalBlock1:
    if (IsNull(P->Code) || !IsCons(P->Code) || !IsCons(Car(P->Code)))
        break;

    if (IsNull(Cdr(P->Code)) || !IsCons(Cadr(P->Code))) {
        Debug("Tail!\n");
        ReplaceState(P, psEvalCommand);
    }
    else {
        SaveState(P, svCode, Cdr(P->Code));
        NewState(P, psEvalCommand);
    }

    P->Code = Car(P->Code);
    P->Scratch = Cdr(P->Scratch);

    /* Continue. */
case psEvalCommand:
    ReplaceState(P, psEvalCommand1);

    SaveState(P, svCode, Cdr(P->Code));

```

```

P->Code = Car(P->Code);

NewState(P, psEvalElement);
continue;

case psEvalCommand1:
    Debug("Evaluating command\n");
    Method = Car(P->Scratch);
    if (IsSymbol(Method))
    {
        ReplaceState(P, psEvalExternal);
        NewState(P, psEvalArguments);
        continue;
    }

    if (!IsMethod(Method))
        break;

    Body = GetMethodBody(Method);

    Debug("Evaluating command: Is it a built-in function?\n");
    if (IsFunction(Body)) {
        Debug("Evaluating command: Yes it is!\n");
        ReplaceState(P, psEvalBuiltIn);
        NewState(P, psEvalArguments);
        continue;
    }
    else if (IsInteger(Body)) {
        Debug("Evaluating command: No it is a state!\n");
        ReplaceState(P, UnwrapInteger(Body));
        continue;
    }

    ReplaceState(P, psEvalApplication);
    NewState(P, psEvalArguments);

    /* Continue. */
case psEvalArguments:
    P->Scratch = Cons(NULL, P->Scratch);

    ReplaceState(P, psEvalArguments1);

    /* Continue. */
case psEvalArguments1:
    if (IsNull(P->Code))
        break;

```

```

SaveState(P, svCode, Cdr(P->Code));

P->Code = Car(P->Code);

NewState(P, psEvalElement);

/* Continue. */
case psEvalElement:
    if (IsCons(P->Code)) {
        if (IsAtom(Cdr(P->Code)))
            ReplaceState(P, psEvalAccess);
        else
            ReplaceState(P, psEvalCommand);

        continue;
    }
    else if (IsSymbol(P->Code)) {
        Debug("Trying to resolve '%s'\n", UnwrapSymbol(P->Code));
        P->Scratch = Cons(Cdr(Resolve(P->Env, P->Scope, P->Code)),
                        P->Scratch);
    }
    else
        P->Scratch = Cons(P->Code, P->Scratch);

    break;

case psEvalAccess:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);

    P->Scratch = Cons(Cdr(P->Code), P->Scratch);
    P->Code = Car(P->Code);

    NewState(P, psEvalAccess1);

    NewState(P, psEvalElement);
    continue;

case psEvalAccess1:
    P->Code = Cadr(P->Scratch);
    P->Env = NULL;
    P->Scope = Car(P->Scratch);
    P->Scratch = Cddr(P->Scratch);

    ReplaceState(P, psEvalElement);

```

```

    continue;

case psEvalApplication:
    Arguments = GetArguments(P);
    Method = Car(P->Scratch);
    Body = GetMethodBody(Method);
    Parameters = GetMethodParameters(Method);
    Self = GetMethodSelf(Method);

    RemoveState(P);
    if (SaveState(P, svEnv | svScope, P->Env, P->Scope))
        P->Env = CreateEnv(P->Env);

    P->Code = Body;
    P->Scope = CreateScope(GetMethodScope(Method));

    while (!IsNull(Parameters)) {
        ExtendScope(P->Scope, Car(Parameters), Car(Arguments), 1);
        Arguments = Cdr(Arguments);
        Parameters = Cdr(Parameters);
    }
    ExtendScope(P->Scope, WrapSymbol("$args"), Arguments, 1);
    ExtendScope(P->Scope, WrapSymbol("$self"), Self, 1);
    BindContinuation(P);

    NewState(P, psEvalBlock1);
    continue;

case psEvalQuoteCommand:
    Car(P->Scratch) = Car(P->Code);
    break;

case psEvalSpliceCommand:
    ReplaceState(P, psEvalSpliceCommand1);

    P->Code = Car(P->Code);
    P->Scratch = Cdr(P->Scratch);

    NewState(P, psEvalElement);
    continue;

case psEvalSpliceCommand1:
    Arguments = Car(P->Scratch);
    P->Scratch = Cdr(P->Scratch);

    while (!IsNull(Arguments)) {

```



```

        P->Scratch = Cons(Car(Arguments), P->Scratch);
        Arguments = Cdr(Arguments);
    }

    break;

case psEvalSetCommand:
    ReplaceState(P, psEvalSetCommand1);
    NewState(P, psEvalArguments);
    SaveState(P, svCode, Cdr(P->Code));

    P->Code = Car(P->Code);

    NewState(P, psEvalReference);

    /* Continue. */
case psEvalReference:
    RemoveState(P);

    P->Scratch = Cdr(P->Scratch);

    if (IsCons(P->Code)) {
        SaveState(P, svScope, P->Scope);
        NewState(P, psEvalReference1);
        SaveState(P, svCode, Cdr(P->Code));

        P->Code = Car(P->Code);

        NewState(P, psEvalReference2);
        NewState(P, psEvalElement);
        continue;
    }

    NewState(P, psEvalReference1);

    /* Continue. */
case psEvalReference1:
    P->Scratch = Cons(Resolve(P->Env, P->Scope, P->Code), P->Scratch);

    if (Caar(P->Scratch) == P->Code && Cdar(P->Scratch) == P->Code) {
        fprintf(stderr, "choc: %s is undefined.\n", ToString(P->Code));
        exit(0);
    }

    break;

case psEvalReference2:

```

```

P->Scope = Car(P->Scratch);
P->Scratch = Cdr(P->Scratch);
break;

case psEvalBuiltIn:
    if (EvalBuiltIn(P))
        break;

/*

Arguments = GetArguments(P);

Method = Car(P->Scratch);
Body = GetMethodBody(Method);

assert((UnwrapFunction(Body))(P, Arguments));

*/

continue;

case psEvalBlockCommand:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);

    P->Env = CreateEnv(P->Env);
    P->Scope = CreateScope(P->Scope);

    NewState(P, psEvalBlock1);
    continue;

case psEvalDefineCommand:
    Self = ClearFlag(GetMethodSelf(Car(P->Scratch)), stPublic);

    Car(P->Scratch) = Self;
    P->Scratch = Cons(Car(P->Code), P->Scratch);
    P->Code = Cadr(P->Code);

    ReplaceState(P, psEvalDefineCommand1);

    if (!IsNull(P->Code)) {
        NewState(P, psEvalElement);
        continue;
    }

    P->Scratch = Cons(P->Code, P->Scratch);

    /* Continue. */
case psEvalDefineCommand1:
    ExtendScope(Caddr(P->Scratch), Cadr(P->Scratch),

```

```

        Car(P->Scratch), 0);

P->Scratch = Cons(Car(P->Scratch), Cddr(P->Scratch));

break;

case psEvalDynamicCommand:
    Car(P->Scratch) = Car(P->Code);
    P->Code = Cadr(P->Code);

    ReplaceState(P, psEvalDynamicCommand1);

    if (!IsNull(P->Code)) {
        NewState(P, psEvalElement);
        continue;
    }

    P->Scratch = Cons(P->Code, P->Scratch);

    /* Continue. */
case psEvalDynamicCommand1:
    ExtendEnv(P->Env, Cadr(P->Scratch), Car(P->Scratch));

    P->Scratch = Cons(Car(P->Scratch), Cddr(P->Scratch));

    break;

case psEvalSpawnCommand:
    ChildProcess = CreateProcess(P->Env, P->Scope);
    ChildProcess->Code = P->Code;

    NewState(ChildProcess, psEvalBlock1);

    pthread_create(&ChildThread, NULL, ThreadFunction, ChildProcess);

    Car(P->Scratch) = (void*) ChildThread;
    break;

case psEvalIfCommand:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);
    NewState(P, psEvalIfCommand1);
    SaveState(P, svCode, Cdr(P->Code));

    P->Code = Car(P->Code);
    P->Env = CreateEnv(P->Env);

```

```

P->Scope = CreateScope(P->Scope);
P->Scratch = Cdr(P->Scratch);

NewState(P, psEvalElement);
continue;

case psEvalIfCommand1:
    if (!ToBoolean(Car(P->Scratch))) {
        while (!IsNull(Car(P->Code)) && !IsAtom(Car(P->Code)))
            P->Code = Cdr(P->Code);
        P->Code = Cdr(P->Code);
    }

    if (IsNull(P->Code)) {
        break;
    }

    ReplaceState(P, psEvalBlock1);
    continue;

case psEvalPublicCommand:
    Self = ClearFlag(GetMethodSelf(Car(P->Scratch)), stPublic);

    Car(P->Scratch) = Self;
    P->Scratch = Cons(Car(P->Code), P->Scratch);
    P->Code = Cadr(P->Code);

    ReplaceState(P, psEvalPublicCommand1);

    if (!IsNull(P->Code)) {
        NewState(P, psEvalElement);
        continue;
    }

    P->Scratch = Cons(P->Code, P->Scratch);

    /* Continue. */
case psEvalPublicCommand1:
    ExtendScope(Caddr(P->Scratch), Cadr(P->Scratch),
                Car(P->Scratch), 1);

    Cdr(P->Scratch) = Cdddr(P->Scratch);
    break;

case psEvalMethodCommand:
    Parameters = Nil;

```

```

while (!IsCons(Car(P->Code))) {
    Parameters = Cons(Car(P->Code), Parameters);
    P->Code = Cdr(P->Code);
}

Body = P->Code;
Parameters = Reverse(Parameters);

Car(P->Scratch) = CreateMethod(P->Scope, Body, Parameters);
break;

case psEvalImportCommand:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);
    NewState(P, psEvalObjectCommand1);

    P->Env = CreateEnv(P->Env);
    P->Scope = CreateScope(P->Scope);

    SaveState(P, svScope, P->Scope);

    ImportFile(P);

    NewState(P, psEvalBlock);
    continue;

case psEvalObjectCommand:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);
    NewState(P, psEvalObjectCommand1);

    P->Env = CreateEnv(P->Env);
    P->Scope = CreateScope(P->Scope);

    NewState(P, psEvalBlock);
    continue;

case psEvalObjectCommand1:
    SetFlag(P->Scope, stPublic);

    Car(P->Scratch) = P->Scope;
    Cdr(P->Scratch) = Cddr(P->Scratch);
    break;

case psEvalSetCommand1:
    Arguments = GetArguments(P);

```

```

    Self = Car(P->Scratch);

    Cdr(Self) = Car(Arguments);
    Car(P->Scratch) = Car(Arguments);

    break;

case psEvalReturnCommand:
    Method = Car(P->Scratch);
    Parameters = GetMethodParameters(Method);

    P->Stack = Cadr(Parameters);
    P->Scratch = Car(Parameters);

    if (!IsNull(P->Code)) {
        P->Code = Car(P->Code);

        NewState(P, psEvalElement);
    }
    else
        P->Scratch = Cons(Nil, P->Scratch);

    continue;

case psEvalForCommand:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);
    NewState(P, psEvalForCommand1);

    P->Scratch = Cdr(P->Scratch);

    NewState(P, psEvalArguments);
    continue;

case psEvalForCommand1:
    ReplaceState(P, psEvalForCommand2);
    Arguments = GetArguments(P);

    P->Scratch = Cons(Cadr(Arguments), P->Scratch);
    P->Scratch = Cons(Car(Arguments), P->Scratch);

    /* Continue. */
case psEvalForCommand2:
    if (IsNull(Car(P->Scratch)))
        break;

```

```

Arguments = Caar(P->Scratch);
Debug(" For: %s\n", ToString(Arguments));

Car(P->Scratch) = Cdar(P->Scratch);
SaveState(P, svScratch, P->Scratch);

P->Scratch = Cdr(P->Scratch);
P->Scratch = Cons(NULL, P->Scratch);
P->Scratch = Cons(Arguments, P->Scratch);
NewState(P, psEvalApplication);
continue;

case psEvalWhileCommand:
    RemoveState(P);
    SaveState(P, svEnv | svScope, P->Env, P->Scope);
    NewState(P, psEvalWhileCommand1);

    /* Continue. */
case psEvalWhileCommand1:
    ReplaceState(P, psEvalWhileCommand2);
    SaveState(P, svCode, P->Code);

    P->Code = Car(P->Code);
    P->Scratch = Cdr(P->Scratch);

    NewState(P, psEvalElement);
    continue;

case psEvalWhileCommand2:
    if (!ToBoolean(Car(P->Scratch)))
        break;

    ReplaceState(P, psEvalWhileCommand1);
    SaveState(P, svCode, P->Code);

    P->Code = Cdr(P->Code);

    NewState(P, psEvalBlock1);
    continue;

case psEvalBacktick:
    Debug(" Evaluating backtick.\n");

    RemoveState(P);
    SaveState(P, svEnv, P->Env);

```

```

ChannelObject = CreateChannelObject(P, 0, 0, 0);

fcntl(GetChannel(ChannelObject)->Pipe[1], F_SETFD, FD_CLOEXEC);

ChildProcess = CreateProcess(P->Env, P->Scope);

ChildProcess->Code = Car(P->Code);
ExtendEnv(ChildProcess->Env, WrapSymbol("$stdout"), ChannelObject);

NewState(ChildProcess, psEvalPipeChild);
NewState(ChildProcess, psEvalCommand);

pthread_create(&ChildThread, NULL, ThreadFunction, ChildProcess);

File = GetChannel(ChannelObject)->ReadEnd;

Index = 0;
while(fgets(Line, sizeof(Line), File)) {
    Debug("Capturing: %s", Line);

    Buffer = strchr(Line, '\n');
    if (Buffer != NULL)
        *Buffer = '\0';

    if (Index++)
        AppendTo(Arguments, WrapString(GC_strdup(Line)), 0);
    else
        Arguments = Cons(WrapString(GC_strdup(Line)), Nil);
}

Debug("Closing stdin (%d)\n", fileno(File));
fclose(File);

pthread_join(ChildThread, NULL);

Car(P->Scratch) = Arguments;
break;

case psEvalPipe:
    Debug("Evaluating pipe.\n");

    RemoveState(P);
    SaveState(P, svEnv, P->Env);

    ChannelObject = CreateChannelObject(P, 0, 0, 0);

```



```

fcntl(GetChannel(ChannelObject)->Pipe[1], F_SETFD, FD_CLOEXEC);

ChildProcess = CreateProcess(P->Env, P->Scope);

ChildProcess->Code = Car(P->Code);
ExtendEnv(ChildProcess->Env, WrapSymbol("$stdout"), ChannelObject);

NewState(ChildProcess, psEvalPipeChild);
NewState(ChildProcess, psEvalCommand);

pthread_create(&ChildThread, NULL, ThreadFunction, ChildProcess);

P->Code = Cadr(P->Code);
P->Env = CreateEnv(P->Env);
Car(P->Scratch) = (void*) ChildThread;

ExtendEnv(P->Env, WrapSymbol("$stdin"), ChannelObject);

NewState(P, psEvalPipeParent);
NewState(P, psEvalCommand);
continue;

case psEvalPipeChild:
    File = GetChannel(GetValue(P->Env, P->Scope, "$stdout"))->WriteEnd;

    Debug("Closing stdout (%d)\n", fileno(File));
    fclose(File);
    break;

case psEvalPipeParent:
    File = GetChannel(GetValue(P->Env, P->Scope, "$stdin"))->ReadEnd;

    Debug("Closing stdin (%d)\n", fileno(File));
    fclose(File);

    pthread_join((pthread_t) Cadr(P->Scratch), NULL);

    Cdr(P->Scratch) = Cddr(P->Scratch);
    break;

case psEvalPipeStderr:
    Debug("Evaluating pipe.\n");

    RemoveState(P);
    SaveState(P, svEnv, P->Env);

```

```

ChannelObject = CreateChannelObject(P, 0, 0, 0);

fcntl(GetChannel(ChannelObject)->Pipe[1], F_SETFD, FD_CLOEXEC);

ChildProcess = CreateProcess(P->Env, P->Scope);

ChildProcess->Code = Car(P->Code);
ExtendEnv(ChildProcess->Env, WrapSymbol("$stderr"), ChannelObject);

NewState(ChildProcess, psEvalPipeStderrChild);
NewState(ChildProcess, psEvalCommand);

pthread_create(&ChildThread, NULL, ThreadFunction, ChildProcess);

P->Code = Cadr(P->Code);
P->Env = CreateEnv(P->Env);
Car(P->Scratch) = (void*) ChildThread;

ExtendEnv(P->Env, WrapSymbol("$stdin"), ChannelObject);

NewState(P, psEvalPipeParent);
NewState(P, psEvalCommand);
continue;

case psEvalPipeStderrChild:
    File = GetChannel(GetValue(P->Env, P->Scope, "$stderr"))->WriteEnd;

    Debug("Closing stdout (%d)\n", fileno(File));
    fclose(File);
    break;

case psEvalRedirectStdin:
    Debug("Evaluating redirection.\n");

    RemoveState(P);
    SaveState(P, svEnv, P->Env);

    P->Env = CreateEnv(P->Env);

    NewState(P, psEvalRedirectStdin2);
    NewState(P, psEvalRedirectStdin1);
    SaveState(P, svCode, P->Code);

    P->Code = Car(P->Code);

```

```

NewState(P, psEvalElement);
continue;

case psEvalRedirectStdin1:
  if (!IsChannel(Car(P->Scratch))) {
    File = fopen(ToString(Car(P->Scratch)), "r");
    Car(P->Scratch) = CreateChannelObject(P, 1, fileno(File), -1);
  }

  Debug(" Redirecting to channel (%p)\n", Car(P->Scratch));

  ExtendEnv(P->Env, WrapSymbol("$stdin", Car(P->Scratch)));

  P->Code = Cdr(P->Code);

  ReplaceState(P, psEvalCommand);
  continue;

case psEvalRedirectStdin2:
  Debug(" Closing file\n");

  Channel = GetChannel(Cadr(P->Scratch));

  if (Channel->IsFile) {
    fclose(Channel->ReadEnd);
    Channel->ReadEnd = NULL;
  }

  Cdr(P->Scratch) = Cdddr(P->Scratch);
  break;

case psEvalRedirectStdout:
  Debug(" Evaluating redirection.\n");

  RemoveState(P);
  SaveState(P, svEnv, P->Env);

  P->Env = CreateEnv(P->Env);

  NewState(P, psEvalRedirectStdout2);
  NewState(P, psEvalRedirectStdout1);
  SaveState(P, svCode, Cdr(P->Code));

  P->Code = Car(P->Code);

  NewState(P, psEvalElement);

```

```

        continue;

    case psEvalRedirectStdout1:
        if (!IsChannel(Car(P->Scratch))) {
            File = fopen(ToString(Car(P->Scratch)), "w");
            Car(P->Scratch) = CreateChannelObject(P, 1, -1, fileno(File));
        }

        Debug(" Redirecting to channel (%p)\n", Car(P->Scratch));

        ExtendEnv(P->Env, WrapSymbol("$stdout"), Car(P->Scratch));

        ReplaceState(P, psEvalCommand);
        continue;

    case psEvalRedirectStdout2:
        Debug(" Closing file\n");

        Channel = GetChannel(Cadr(P->Scratch));

        if (Channel->IsFile) {
            fclose(Channel->WriteEnd);
            Channel->WriteEnd = NULL;
        }

        Cdr(P->Scratch) = Cdddr(P->Scratch);
        break;

    case psEvalRedirectStderr:
        Debug(" Evaluating redirection.\n");

        RemoveState(P);
        SaveState(P, svEnv, P->Env);

        P->Env = CreateEnv(P->Env);

        NewState(P, psEvalRedirectStderr2);
        NewState(P, psEvalRedirectStderr1);
        SaveState(P, svCode, P->Code);

        P->Code = Car(P->Code);

        NewState(P, psEvalElement);
        continue;

    case psEvalRedirectStderr1:

```

```

    if (!IsChannel(Car(P->Scratch))) {
        File = fopen(ToString(Car(P->Scratch)), "w");
        Car(P->Scratch) = CreateChannelObject(P, 1, -1, fileno(File));
    }

    Debug(" Redirecting to channel (%p)\n", Car(P->Scratch));

    ExtendEnv(P->Env, WrapSymbol("$stderr"), Car(P->Scratch));

    P->Code = Cdr(P->Code);

    ReplaceState(P, psEvalCommand);
    continue;

case psEvalRedirectStderr2:
    Debug(" Closing file\n");

    Channel = GetChannel(Cadr(P->Scratch));

    if (Channel->IsFile) {
        fclose(Channel->WriteEnd);
        Channel->WriteEnd = NULL;
    }

    Cdr(P->Scratch) = Cdddr(P->Scratch);
    break;

case psEvalRedirectAppendStdout:
    Debug(" Evaluating redirection.\n");

    RemoveState(P);
    SaveState(P, svEnv, P->Env);

    P->Env = CreateEnv(P->Env);

    NewState(P, psEvalRedirectStdout2);
    NewState(P, psEvalRedirectAppendStdout1);
    SaveState(P, svCode, P->Code);

    P->Code = Car(P->Code);

    NewState(P, psEvalElement);
    continue;

case psEvalRedirectAppendStdout1:
    if (!IsChannel(Car(P->Scratch))) {

```

```

        File = fopen(ToString(Car(P->Scratch)), "a");
        Car(P->Scratch) = CreateChannelObject(P, 1, -1, fileno(File));
    }

    Debug("Redirecting to channel (%p)\n", Car(P->Scratch));

    ExtendEnv(P->Env, WrapSymbol("$stdout"), Car(P->Scratch));

    P->Code = Cdr(P->Code);

    ReplaceState(P, psEvalCommand);
    continue;

case psEvalRedirectAppendStderr:
    Debug("Evaluating redirection.\n");

    RemoveState(P);
    SaveState(P, svEnv, P->Env);

    P->Env = CreateEnv(P->Env);

    NewState(P, psEvalRedirectStderr2);
    NewState(P, psEvalRedirectAppendStderr1);
    SaveState(P, svCode, P->Code);

    P->Code = Car(P->Code);

    NewState(P, psEvalElement);
    continue;

case psEvalRedirectAppendStderr1:
    if (!IsChannel(Car(P->Scratch))) {
        File = fopen(ToString(Car(P->Scratch)), "a");
        Car(P->Scratch) = CreateChannelObject(P, 1, -1, fileno(File));
    }

    Debug("Redirecting to channel (%p)\n", Car(P->Scratch));

    ExtendEnv(P->Env, WrapSymbol("$stderr"), Car(P->Scratch));

    P->Code = Cdr(P->Code);

    ReplaceState(P, psEvalCommand);
    continue;

case psEvalExternal:

```

```

        Arguments = GetArguments(P);

        Debug(" External command '%s' .\n", UnwrapText( Car(P->Scratch)));

        assert( External(P, Arguments));

        break;

    default:
        if (State < svMax)
            RestoreState(P);

        break;
    }

    RemoveState(P);
}

return NULL;
}

/*****

    Built-in Functions.

*****/

#define Generator(Name, Type, Default) \
    int Fresh ## Name(void* Process, void* Arguments) { \
        process* P = (process*) Process; \
        \
        Type Value = Default; \
        Debug(" Inside %s\n", #Name); \
        if (!IsNull(Arguments)) \
            Value = To ## Name(Car(Arguments)); \
        \
        Car(P->Scratch) = Wrap ## Name(Value); \
        \
        return 1; \
    }

Generator(Boolean, int, 0)
Generator(Double, double, 0.0)
Generator(Integer, int, 0)
Generator(Status, int, 0)

```

```

static int FreshChannel(void* Process, void* Arguments) {
    process* P = (process*) Process;
    void* ChannelObject = CreateChannelObject(P, 0, 0, 0);

    Car(P->Scratch) = ChannelObject;

    return 1;
}

#define ArithmeticOperator(Name, Type, Op) \
    int Name(void* Process, void* Arguments) { \
        process* P = (process*) Process; \
        int Total = 0; \
        \
        Total = To ## Type(Car(Arguments)); \
        \
        Arguments = Cdr(Arguments); \
        while (!IsNull(Arguments)) { \
            Total = Total Op To ## Type(Car(Arguments)); \
            \
            Arguments = Cdr(Arguments); \
        } \
        \
        Car(P->Scratch) = Wrap ## Type(Total); \
        \
        return 1; \
    }

ArithmeticOperator(Add, Integer, +)
ArithmeticOperator(Subtract, Integer, -)
ArithmeticOperator(Multiply, Integer, *)
ArithmeticOperator(Divide, Integer, /)
ArithmeticOperator(Modulo, Integer, %)

/* For now -and and -or do not short circuit. */
int And(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int Value = 0;

    do {
        Value = ToBoolean(Car(Arguments));
        Arguments = Cdr(Arguments);
    } while (Value && !IsNull(Arguments));

    Car(P->Scratch) = WrapBoolean(Value);

```



```

    return 1;
}

int Or(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int Value = 0;

    do {
        Value = ToBoolean(Car(Arguments));
        Arguments = Cdr(Arguments);
    } while (!Value && !IsNull(Arguments));

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

int Equivalent(void* Process, void* Arguments) {
    process* P = (process*) Process;
    void* Cell = Car(Arguments);

    int Value = 1;
    Arguments = Cdr(Arguments);
    while (Value && !IsNull(Car(Arguments))) {
        Value = Equal(Cell, Car(Arguments));

        Arguments = Cdr(Arguments);
    }

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

int Identical(void* Process, void* Arguments) {
    process* P = (process*) Process;
    void* Cell = Car(Arguments);

    int Value = 1;
    Arguments = Cdr(Arguments);
    while (Value && !IsNull(Car(Arguments))) {
        Value = Cell == Car(Arguments);

        Arguments = Cdr(Arguments);
    }

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

```

```

}

static int Not(void* Process, void* Arguments) {
    process* P = (process*) Process;

    Car(P->Scratch) = WrapBoolean(!ToBoolean(Car(Arguments)));

    return 1;
}

/*
 * This only tests to make sure that the first argument is not equivalent
 * to the rest. A better version might check to make sure that none of the
 * arguments are equivalent.
 */
int NotEquivalent(void* Process, void* Arguments) {
    process* P = (process*) Process;
    void* Cell = Car(Arguments);

    int Value = 1;
    Arguments = Cdr(Arguments);
    while (Value && !IsNull(Car(Arguments))) {
        Value = !Equal(Cell, Car(Arguments));

        Arguments = Cdr(Arguments);
    }

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

int NotLess(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int Current = ToInteger(Car(Arguments));

    int Value = 1;
    Arguments = Cdr(Arguments);
    while (Value && !IsNull(Car(Arguments))) {
        int Next = ToInteger(Car(Arguments));
        Value = Current >= Next;

        Current = Next;
        Arguments = Cdr(Arguments);
    }
}

```

```

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

int Greater(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int Current = ToInteger(Car(Arguments));

    int Value = 1;
    Arguments = Cdr(Arguments);
    while (Value && !IsNull(Car(Arguments))) {
        int Next = ToInteger(Car(Arguments));
        Value = Current > Next;

        Current = Next;
        Arguments = Cdr(Arguments);
    }

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

int NotGreater(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int Current = ToInteger(Car(Arguments));

    int Value = 1;
    Arguments = Cdr(Arguments);
    while (Value && !IsNull(Car(Arguments))) {
        int Next = ToInteger(Car(Arguments));
        Value = Current <= Next;

        Current = Next;
        Arguments = Cdr(Arguments);
    }

    Car(P->Scratch) = WrapBoolean(Value);
    return 1;
}

int Less(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int Current = ToInteger(Car(Arguments));

    int Value = 1;

```

```

Arguments = Cdr(Arguments);
while (Value && !IsNull(Car(Arguments))) {
    int Next = ToInteger(Car(Arguments));
    Value = Current < Next;

    Current = Next;
    Arguments = Cdr(Arguments);
}

Car(P->Scratch) = WrapBoolean(Value);
return 1;
}

static char* Chdir(process* P, char* Arg) {
    char* Dir = UnwrapString(GetValue(P->Env, P->Scope, "$cwd"));

    if (*Arg != '/') {
        int ArgLen = strlen(Arg);
        int DirLen = strlen(Dir);
        char* Temp = GC_malloc_atomic(ArgLen + DirLen + 2);

        memcpy(Temp, Dir, DirLen);
        Temp[DirLen] = '/';
        memcpy(Temp + DirLen + 1, Arg, ArgLen + 1);

        Arg = Temp;
    }

    Dir = GC_malloc_atomic(PATHMAX);
    if (realpath(Arg, Dir) == NULL || P != MainProcess || chdir(Dir))
        return NULL;

    return Dir;
}

static int ApplyCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;

    void* Method = Car(Arguments);
    Arguments = Cdr(Arguments);

    if (!IsMethod(Method)) {
        Car(P->Scratch) = False;
        return 1;
    }
}

```

```

    Car(P->Scratch) = Method;
    P->Scratch = Cons(NULL, P->Scratch);
    while (!IsNull(Arguments)) {
        P->Scratch = Cons(Car(Arguments), P->Scratch);
        Arguments = Cdr(Arguments);
    }

    ReplaceState(P, psEvalApplication);

    return 0;
}

static int CdCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;
    char* Arg = ToString(Car(Arguments));

    char* Dir = Chdir(P, Arg);

    if (Dir != NULL) {
        Car(P->Scratch) = WrapSymbol(Dir);
        ExtendEnv(P->Env, WrapSymbol("$cwd"), Car(P->Scratch));
    }
    else
        Car(P->Scratch) = False;

    return 1;
}

static int ChildCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;
    void* Object = ClearFlag(GetMethodSelf(Car(P->Scratch)), stPublic);

    if (!IsObject(Object)) {
        Car(P->Scratch) = False;
        return 1;
    }

    Object = CreateScope(Object);
    SetFlag(Object, stPublic);

    Car(P->Scratch) = Object;

    return 1;
}

static int CloneCommand(void* Process, void* Arguments) {

```

```

process* P = (process*) Process;
void* Object = ClearFlag(GetMethodSelf(Car(P->Scratch)), stPublic);

if (!IsObject(Object)) {
    Car(P->Scratch) = False;
    return 1;
}

Object = CopyObject(Object);
SetFlag(Object, stPublic);

Car(P->Scratch) = Object;

return 1;
}

static int ConsCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;

    Car(P->Scratch) = Cons(Car(Arguments), Cadr(Arguments));

    return 1;
}

#define ConsCellAccess(Type) \
    static int C ## Type ## rCommand(void* Process, void* Arguments) { \
        process* P = (process*) Process; \
        \
        Car(P->Scratch) = C ## Type ## r(Car(Arguments)); \
        \
        return 1; \
    }

ConsCellAccess(a)
ConsCellAccess(d)
ConsCellAccess(aa)
ConsCellAccess(ad)
ConsCellAccess(da)
ConsCellAccess(dd)
ConsCellAccess(aaa)
ConsCellAccess(aad)
ConsCellAccess(ada)
ConsCellAccess(add)
ConsCellAccess(daa)
ConsCellAccess(dad)
ConsCellAccess(dda)

```

```

ConsCellAccess(ddd)

#define CellType(Type, Unused1, Unused2) \
    static int Is ## Type ## Command(void* Process, void* Arguments) { \
        process* P = (process*) Process; \
        \
        Car(P->Scratch) = WrapBoolean(Is ## Type(Car(Arguments))); \
        \
        return 1; \
    }
CellType(Atom, Unused, "atom")
CellType(Boolean, Unused, "boolean")
CellType(Cons, Unused, "cons")
CellType(List, Unused, "list")
CellType(Null, Unused, "null")
CellType(Object, Unused, "object")
CellType(Text, Unused, "text")
#include "cell.def"

static int EchoCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;

    void* ChannelObject = GetValue(P->Env, P->Scope, "$stdout");
    void* WriteMethod = GetValue(Null, ChannelObject, "write");

    /* Replace echo method with stdout's write method. */
    Car(P->Scratch) = WriteMethod;

    assert((UnwrapFunction(GetMethodBody(WriteMethod)))(P, Arguments));

    return 1;
}

static int ExitCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;
    int ExitStatus = 0;

    if (!IsNull(Arguments))
        ExitStatus = ToInteger(Car(Arguments));

    Car(P->Scratch) = WrapInteger(ExitStatus);

    exit(ExitStatus);

    return 1;
}

```

```

static int IncludeCommand(void* Process, void* Arguments) {
    char Buffer[1024];

    FILE* File = NULL;
    parser* Parser = NULL;
    process* P = (process*) Process;
    scanner* Scanner = NULL;

    void* Command = Nil;
    void* SavedStack = P->Stack;
    char* Name = ToString(Car(Arguments));

    Debug("Trying to include %s\n", Name);

    File = fopen(Name, "r");
    if (File == NULL) {
        Car(P->Scratch) = False;
        return 1;
    }

    Parser = CreateParser(&Command, SetCommand);
    Scanner = CreateScanner(Parser, Parse);

    P->Stack = Nil;

    while (fgets(Buffer, sizeof(Buffer), File) != NULL) {
        Debug("Read %s\n", Buffer);

        Scan(Scanner, Buffer);
        if (IsNull(Command))
            continue;

        Debug("Executing %s\n", ToString(Car(Command)));

        P->Code = Command;
        NewState(P, psEvalCommand);

        ThreadFunction(P);

        P->Scratch = Cdr(P->Scratch);
        Command = Nil;
    }

    fclose(File);

```



```

    P->Stack = SavedStack;
    Car(P->Scratch) = True;

    return 1;
}

static int ImportFile(process* P) {
    char Buffer[1024];

    FILE* File = NULL;
    parser* Parser = NULL;
    scanner* Scanner = NULL;

    void* Block = Nil;
    void* Command = Nil;

    char* Name = ToString(Car(P->Code));
    int Index = 0;

    Debug("Trying to import %s\n", Name);

    File = fopen(Name, "r");
    if (File == NULL) {
        P->Code = Block;
        return 1;
    }

    Parser = CreateParser(&Command, SetCommand);
    Scanner = CreateScanner(Parser, Parse);

    while (fgets(Buffer, sizeof(Buffer), File) != NULL) {
        Debug("Read %s\n", Buffer);

        Scan(Scanner, Buffer);
        if (IsNull(Command))
            continue;

        if (Index++)
            AppendTo(Block, Command, 0);
        else
            Block = Cons(Command, Nil);

        Command = Nil;
    }

    fclose(File);
}

```

```

    P->Code = Block;

    return 1;
}

static int ReadCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;

    void* ChannelObject = GetValue(P->Env, P->Scope, "$stdin");
    void* ReadMethod = GetValue(nil, ChannelObject, "read");

    /* Replace read method with stdin's read method. */
    Car(P->Scratch) = ReadMethod;

    assert((UnwrapFunction(GetMethodBody(ReadMethod)))(P, Arguments));

    return 1;
}

static int ReadlineCommand(void* Process, void* Arguments) {
    char Line[1024];

    char* Buffer = NULL;
    process* P = (process*) Process;
    void* ChannelObject = GetValue(P->Env, P->Scope, "$stdin");
    channel* Channel = GetChannel(ChannelObject);

    Car(P->Scratch) = Nil;
    if (fgets(Line, sizeof(Line), Channel->ReadEnd) == NULL)
        return 1;

    if (*Line == '\0')
        return 1;

    Buffer = strrchr(Line, '\n');
    if (Buffer != NULL)
        *Buffer = '\0';

    Car(P->Scratch) = WrapString(GC_strdup(Line));

    return 1;
}

static int WaitCommand(void* Process, void* Arguments) {
    pthread_t Thread;

```

```

    process* P = (process*) Process;

    Thread = (pthread_t) Car(Arguments);

    pthread_join(Thread, NULL);

    Car(P->Scratch) = Nil;

    return 1;
}

static int SetCarCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;

    void* Cell = Car(Arguments);
    void* Value = Cadr(Arguments);

    int Status = IsCons(Cell);

    Car(P->Scratch) = WrapBoolean(Status);

    Car(Cell) = Value;

    return 1;
}

static int SetCdrCommand(void* Process, void* Arguments) {
    process* P = (process*) Process;

    void* Cell = Car(Arguments);
    void* Value = Cadr(Arguments);

    int Status = IsCons(Cell);

    Car(P->Scratch) = WrapBoolean(Status);

    Cdr(Cell) = Value;

    return 1;
}

/*****

Interface Functions.

```

```

*****/

void EngineInit(void) {
    ModifiesGlobal(Settings);

    void* TopLevelEnv = Nil;
    void* TopLevelScope = Nil;

    if (MainProcess != NULL)
        return;

    /* Initialize cell library (also initializes the garbage collector). */
    CellInit(
#define CellType(Name, Unused1, Unused2) \
        ct ## Name, pthread_mutex_lock, pthread_mutex_unlock, & Name ## Mutex,
#include "cell.def"
        0);

    TopLevelEnv = CreateEnv(Nil);
    TopLevelScope = CreateScope(Nil);
    MainProcess = CreateProcess(TopLevelEnv, TopLevelScope);

    /* Use line buffering even if we're not connected to a tty. */
    setlinebuf(stdin);
    setlinebuf(stdout);

    ExtendEnv(TopLevelEnv, WrapSymbol("$stdin"),
        CreateChannelObject(MainProcess, 0, STDIN_FILENO, -1));
    ExtendEnv(TopLevelEnv, WrapSymbol("$stdout"),
        CreateChannelObject(MainProcess, 0, -1, STDOUT_FILENO));
    ExtendEnv(TopLevelEnv, WrapSymbol("$stderr"),
        CreateChannelObject(MainProcess, 0, -1, STDERR_FILENO));
    ExtendEnv(TopLevelEnv, WrapSymbol("$cwd"),
        WrapSymbol(GetWorkingDir()));

    Debug("svMax = %d\n", svMax);

    /* Primitives. */
    BindState(TopLevelScope, psEvalBlockCommand, "block", 0);
    BindState(TopLevelScope, psEvalDefineCommand, "define", 0);
    BindState(TopLevelScope, psEvalDynamicCommand, "dynamic", 0);
    BindState(TopLevelScope, psEvalForCommand, "for", 0);
    BindState(TopLevelScope, psEvalIfCommand, "if", 0);
    BindState(TopLevelScope, psEvalImportCommand, "import", 0);
    BindState(TopLevelScope, psEvalMethodCommand, "method", 0);
    BindState(TopLevelScope, psEvalObjectCommand, "object", 0);

```

```

BindState(TopLevelScope, psEvalQuoteCommand, "quote", 0);
BindState(TopLevelScope, psEvalSetCommand, "set", 0);
BindState(TopLevelScope, psEvalSpawnCommand, "spawn", 0);
BindState(TopLevelScope, psEvalSpliceCommand, "splice", 0);
BindState(TopLevelScope, psEvalWhileCommand, "while", 0);

BindState(TopLevelScope, psEvalPublicCommand, "public", 1);

BindState(TopLevelScope, psEvalSpawnCommand, "op&", 0);
BindState(TopLevelScope, psEvalBacktick, "backtick", 0);
BindState(TopLevelScope, psEvalPipe, "op|", 0);
BindState(TopLevelScope, psEvalPipeStderr, "op!|", 0);
BindState(TopLevelScope, psEvalRedirectStdin, "op<", 0);
BindState(TopLevelScope, psEvalRedirectStdout, "op>", 0);
BindState(TopLevelScope, psEvalRedirectStderr, "op!>", 0);
BindState(TopLevelScope, psEvalRedirectAppendStdout, "op>>", 0);
BindState(TopLevelScope, psEvalRedirectAppendStderr, "op!>>", 0);

/* Generators. */
BindFunction(TopLevelScope, FreshBoolean, "boolean", 0);
BindFunction(TopLevelScope, FreshChannel, "channel", 0);
BindFunction(TopLevelScope, FreshDouble, "double", 0);
BindFunction(TopLevelScope, FreshInteger, "integer", 0);
BindFunction(TopLevelScope, FreshStatus, "status", 0);

/* Operators. */
BindFunction(TopLevelScope, Not, "op!", 0);
BindFunction(TopLevelScope, Add, "op+", 0);
BindFunction(TopLevelScope, Subtract, "op-", 0);
BindFunction(TopLevelScope, And, "op-and", 0);
BindFunction(TopLevelScope, Divide, "op-div", 0);
BindFunction(TopLevelScope, Equivalent, "op-eq", 0);
BindFunction(TopLevelScope, NotLess, "op-ge", 0);
BindFunction(TopLevelScope, Greater, "op-gt", 0);
BindFunction(TopLevelScope, Identical, "op-is", 0);
BindFunction(TopLevelScope, NotGreater, "op-le", 0);
BindFunction(TopLevelScope, Less, "op-lt", 0);
BindFunction(TopLevelScope, Modulo, "op-mod", 0);
BindFunction(TopLevelScope, Multiply, "op-mul", 0);
BindFunction(TopLevelScope, NotEquivalent, "op-ne", 0);
BindFunction(TopLevelScope, Or, "op-or", 0);

/* Built-in Functions. */
BindConsCellAccess(a);
BindConsCellAccess(d);
BindConsCellAccess(aa);

```

```

BindConsCellAccess(ad);
BindConsCellAccess(da);
BindConsCellAccess(dd);
BindConsCellAccess(aaa);
BindConsCellAccess(aad);
BindConsCellAccess(ada);
BindConsCellAccess(add);
BindConsCellAccess(daa);
BindConsCellAccess(dad);
BindConsCellAccess(dda);
BindConsCellAccess(ddd);
BindFunction(TopLevelScope, ConsCommand, "cons", 0);

#define CellType(Type, Unused, Name) \
    BindFunction(TopLevelScope, Is ## Type ## Command, "is-" Name, 0);
    CellType(Atom, Unused, "atom")
    CellType(Boolean, Unused, "boolean")
    CellType(Cons, Unused, "cons")
    CellType(List, Unused, "list")
    CellType(Null, Unused, "null")
    CellType(Object, Unused, "object")
    CellType(Text, Unused, "text")
#include "cell.def"

BindFunction(TopLevelScope, ApplyCommand, "apply", 0);
BindFunction(TopLevelScope, CdCommand, "cd", 0);
BindFunction(TopLevelScope, EchoCommand, "echo", 0);
BindFunction(TopLevelScope, ExitCommand, "exit", 0);
BindFunction(TopLevelScope, IncludeCommand, "include", 0);
BindFunction(TopLevelScope, ReadCommand, "read", 0);
BindFunction(TopLevelScope, ReadlineCommand, "readline", 0);
BindFunction(TopLevelScope, SetCarCommand, "set-car", 0);
BindFunction(TopLevelScope, SetCdrCommand, "set-cdr", 0);
BindFunction(TopLevelScope, WaitCommand, "wait", 0);

BindFunction(TopLevelScope, ChildCommand, "child", 1);
BindFunction(TopLevelScope, CloneCommand, "clone", 1);

ExtendScope(TopLevelScope, WrapSymbol("false"), False, 0);
ExtendScope(TopLevelScope, WrapSymbol("true"), True, 0);

Debug("ArgumentCount = %d\n", Settings->ArgumentCount);

if (Settings->ArgumentCount >= 2)
    Settings->IsScript = 1;

```

```

Debug("IsScript = %d\n", Settings->IsScript);

ExtendScope(TopLevelScope, WrapSymbol("$0"),
            WrapString(Settings->ArgumentList[Settings->IsScript]), 0);

{
    void* Args = Nil;
    if (Settings->IsScript) {
        int Index = Settings->ArgumentCount - 1;
        while (Index >= 2) {
            Args = Cons(WrapString(Settings->ArgumentList[Index--]), Args);
        }
    }

    ExtendScope(TopLevelScope, WrapSymbol("$args"), Args, 0);
}

ExtendScope(TopLevelScope, WrapSymbol("$root"), TopLevelScope, 0);

return;
}

int ProcessCommand(void* Unused, void* Command) {
    MainProcess->Code = Command;
    NewState(MainProcess, psEvalCommand);

    ThreadFunction(MainProcess);

    MainProcess->Scratch = Cdr(MainProcess->Scratch);

    return 0;
}

```

Listing B.20: engine.c

B.7 Interface

```
/* See license.txt for copyright and licensing details. */

#ifndef InterfaceH
#define InterfaceH

int Interface(void*, int (*)(void*, char*));

#endif /* InterfaceH */
```

Listing B.21: interface.h

```
/* See license.txt for copyright and licensing details. */

#include <ctype.h>
#include <libtecla.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

#include "interface.h"
#include "logging.h"
#include "settings.h"

static CPLMATCHFN(CustomCompletion) {
    const char* Start = line;
    int Length = word_end;

    while(*Start && Length--) {
        if (isspace(*Start)) {
            Start++;
            continue;
        }

        if (line[word_end] && !isspace(line[word_end]))
            return 0;

        return cpl_file_completions(cpl, NULL, line, word_end);
    }

    cpl_add_completion(cpl, line, word_end, word_end, "\t", NULL, NULL);

    return 0;
}
```



```

int Script(void* ProcessLineData, int (*ProcessLine)(void*, char*)) {
    UsesGlobal(Settings);

    FILE* Script = NULL;
    char Line[1024];

    Script = fopen(Settings->ArgumentList[1], "r");
    if (Script == NULL)
        return 1;

    while(fgets(Line, sizeof(Line), Script))
        ProcessLine(ProcessLineData, Line);

    fclose(Script);

    return 0;
}

int Interactive(void* ProcessLineData, int (*ProcessLine)(void*, char*)) {
    UsesGlobal(Settings);

    GetLine* LineReader = new_GetLine(1024, 16384);

    gl_customize_completion(LineReader, NULL, CustomCompletion);

    setlocale(LC_CTYPE, "");

    for (;;) {
        char* Line = gl_get_line(LineReader, Settings->Prompt, NULL, 0);

        if (Line != NULL) {
            ProcessLine(ProcessLineData, Line);
        } else {
            break;
        }
    }

    LineReader = del_GetLine(LineReader);

    return 0;
}

int Interface(void* ProcessLineData, int (*ProcessLine)(void*, char*)) {
    UsesGlobal(Settings);

```

```

    if (Settings->IsScript)
        return Script(ProcessLineData, ProcessLine);

    return Interactive(ProcessLineData, ProcessLine);
}

```

Listing B.22: interface.c

```

/* See license.txt for copyright and licensing details. */

#include <stdlib.h>

#include "engine.h"
#include "interface.h"
#include "logging.h"
#include "parser.h"
#include "scanner.h"
#include "settings.h"

int main(int argc, char** argv)
{
    parser* Parser = NULL;
    scanner* Scanner = NULL;

    ModifiesGlobal(Settings);

    Settings->ArgumentCount = argc;
    Settings->ArgumentList = argv;
    Settings->HistorySize = 1000;
    Settings->LineEditingStyle = "emacs";
    Settings->Prompt = "; ";

    SetLogLevel(llNone);

    EngineInit();

    Parser = CreateParser(NULL, ProcessCommand);
    Scanner = CreateScanner(Parser, Parse);

    return Interface(Scanner, Scan);
}

```

Listing B.23: main.c

Bibliography

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, Secaucus, NJ, USA, 1996.
- [2] R. Allen. SELF, 2010. <http://selflanguage.org/>.
- [3] G. S. Blair, J. R. Malone, and J. A. Mariani. A Critique of UNIX. *Software: Practice and Experience*, 15(12):1125–1139, 1985.
- [4] S. R. Bourne. An Introduction to the UNIX Shell. *Bell System Technical Journal*, 57(6):1971–1990, 1978.
- [5] J. Bunting. What is the most efficient way to clone a JavaScript object?, 2008. <http://stackoverflow.com/questions/122102/what-is-the-most-efficient-way-to-clone-a-javascript-object>.
- [6] L. M. Campbell and M. D. Campbell. An Overview of the Ada Shell. In *USENIX Winter*, pages 302–313, 1986.
- [7] S. J. Chapin and R. Stansifer. Using ML as a Command Language. *Software: Practice and Experience*, 20(7):649–659, 1990.
- [8] C. Dony, J. Malenfant, and P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validation. *SIGPLAN Notices*, 27(10):201–217, 1992.
- [9] T. Duff. Rc — A Shell for Plan 9 and UNIX Systems. In *Proceedings of the Summer UKUUG Conference*, pages 21–33, 1990.
- [10] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1987.
- [11] T. Ehrhard and O. Laurent. Interpreting a Finitary Pi-Calculus in Differential Interaction Nets. In *CONCUR*, pages 333–348, 2007.
- [12] J. R. Ellis. A Lisp Shell. *SIGPLAN Notices*, 15(5):24–34, 1980.
- [13] C. W. Fraser and D. R. Hanson. A High-Level Programming and Command Language. *SIGPLAN Notices*, 18(6):212–219, 1983.
- [14] D. Gudeman. Representing Type Information in Dynamically-Typed Languages. Technical Report TR93-27, University of Arizona, Department of Computer Science, Tucson, Arizona, 1993.

- [15] P. Haahr and B. Rakitzis. Es: A Shell with Higher-Order Functions. In *USENIX Winter*, pages 51–60, 1993.
- [16] A. Heller and J. A. Tov. Caml-Shcaml: An OCaml Library for Unix Shell Programming. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 79–90, New York, NY, USA, 2008. ACM.
- [17] J. Karssenberg. Zoidberg - A Modular Perl Shell, 2005. <http://search.cpan.org/~pardus/Zoidberg-0.96/lib/Zoidberg.pm>.
- [18] J. Karssenberg. Zoidberg - A Shell That Speaks Perl, 2005. <http://www.drdoobs.com/architecture-and-design/184416203>.
- [19] R. Kelsey, W. Clinger, J. Rees, H. Abelson, N. I. A. IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. S. Jr., G. J. Sussman, and M. Wand. Revised⁵ Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.
- [20] B. W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, 1983.
- [21] D. Korn. KSH — A Shell Programming Language. In *USENIX Summer*, pages 191–202, 1983.
- [22] D. Korn. KornShell Overview, 2009. <http://www.kornshell.com/info/>.
- [23] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, Providence, RI, version as of 2006-01-15 — 1st version: 2003-12-15 edition, 2006. <http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/PDF/plai-2006-01-15.pdf> – last visited at May 10th 2006.
- [24] S. Maschek. PWB Shell Manual, 2008. <http://www.in-ulm.de/~mascheck/>.
- [25] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1959.
- [26] C. S. McDonald. fsh — A Functional UNIX Command Interpreter. *Software: Practice and Experience*, 17(10):685–700, 1987.
- [27] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, New York, NY, USA, 1999.
- [28] J. A. Neitzel. V6 Thompson Shell Port - Home, 2010. <http://v6shell.org/>.

- [29] A. Oakley and J. Snover. *Monad: Introducing the New MSH Command Shell and Language for Windows*. O'Reilly Media, Inc., 2005.
- [30] J. K. Ousterhout. Tcl: An Embeddable Command Language. In *USENIX*, pages 133–146, 1990.
- [31] F. Pérez and B. E. Granger. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering.*, 9(3):21–29, May 2007.
- [32] B. C. Pierce. Programming in the Pi-calculus: A Tutorial Introduction to Pict. <http://www.dmi.unict.it/~barba/PROG-LANG/PROGRAMMI-TESTI/READING-MATERIAL/TutorialPICT.pdf>, 1997.
- [33] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
- [34] R. Pike. Newsqueak: A Language for Communicating with Mice. Technical Report CSTR 143, Bell Labs, 1989.
- [35] R. Pike. The Implementation of Newsqueak. *Software: Practice and Experience*, 20(7):649–659, 1990.
- [36] G. Purdy. The Perl Shell (psh), 2007. <http://www.focusresearch.com/gregor/sw/psh/>.
- [37] C. Queinnec and K. Callaway. *Lisp in Small Pieces*. Cambridge University Press, New York, NY, USA, 2003.
- [38] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 1999.
- [39] O. Shivers. A Scheme Shell. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1994.
- [40] L. A. Stein, H. Lieberman, and D. Ungar. A Shared View of Sharing: The Treaty of Orlando. In *Object-Oriented Concepts, Databases, and Applications*, pages 31–48. ACM, 1989.
- [41] A. Taivalsaari. Classes vs. Prototypes - Some Philosophical and Historical Observations. In *Journal of Object-Oriented Programming*, pages 44–50. Springer-Verlag, 1996.
- [42] S. A. Uhler. MTX — A Shell that Permits Dynamic Rearrangement of Process Connections and Windows. In *USENIX*, pages 275–285, 1990.

- [43] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, 1991.
- [44] A. Wiggins. Rush, 2008. <http://rush.heroku.com/>.
- [45] N. Wilcox. Pysh: A Python Shell, 2006. <http://pysh.sourceforge.net/>.
- [46] Windows Powershell vs. bash, 2008. <http://fixunix.com/linux/553555-windows-powershell-vs-bash-examples.html>.
- [47] Bash (Unix Shell), 2010. [http://en.wikipedia.org/wiki/Bash_\(Unix_shell\)](http://en.wikipedia.org/wiki/Bash_(Unix_shell)).
- [48] Bourne shell, 2010. http://en.wikipedia.org/wiki/Bourne_shell.
- [49] Comparison of command shells, 2010. http://en.wikipedia.org/wiki/Comparison_of_command_shells.
- [50] *Core JavaScript 1.5 Guide*, 2010. https://developer.mozilla.org/en/Core_JavaScript_1.5_Guide.
- [51] PWB shell, 2010. http://en.wikipedia.org/wiki/PWB_shell.
- [52] Z Shell, 2010. <http://en.wikipedia.org/wiki/Zsh>.