Department of Computer Science
University of Canterbury,
Christchurch, New Zealand

COSC460 2016

Dissertation

Mary Had a Little Lambda: Implementing a Minimal Lisp for Assisting with Education

Author:
Andrew Bell

Supervisor:
Dr. Kourosh Neshatian

October 14, 2016

Abstract

In this report, we describe the implementation of a minimal version of Lisp for use in teaching programmers who have some experience in another language. We discuss the reasons why learning Lisp is beneficial, and what such a minimal language might achieve. After mentioning other Lisp dialects which are minimal, we explain why these do not meet the needs we identify. We then report on the design and philosophy of our newly created language, Mary, and the most important features of this dialect. This language is placed in the context of other research and we report on the extent to which Mary meets our design goals.

"Lisp isn't a language, it's a building material."

Alan Kay [6]

"Lisp is worth learning for ... the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."

Eric Raymond [25]

"Stand by the roads, and look, and ask for the ancient paths, where the good way is; and walk in it."

Jeremiah 6:16, *ESV*

Acknowledgements

Thank you to my family and friends for your support this year. Especially to Joshua and Dannielle for proof reading this.

Thank you to Dr. Kourosh Neshatian for the initial idea for this work, which I have thoroughly enjoyed, and for showing a genuine care for both the project and my wellbeing throughout the year.

S. D. G.

Contents

Chapter 1

Introduction

This report describes the design, philosophy and implementation of a minimal Lisp for the purposes of education. This language has been created for use at tertiary level in order to teach Lisp to students who may know one or two other languages, but are perhaps unfamiliar with functional programming and related concepts.

Lisp is one of the oldest languages still in common usage, and has had a far-reaching influence on the development of many other languages in both explicit and subtle ways. It has spawned many different dialects with particular strengths and weaknesses, most notably Common Lisp, Scheme and Clojure [22]. Another area where Lisp has been especially influential is the development of high level, dynamically typed and interpreted languages such as Perl, Ruby and Python, and functional languages such as Haskell and Elixir [5, 13].

When John McCarthy first developed Lisp in the years between 1956-1962, one of his key discoveries was that with only seven key functions defined in Lisp, one could create an `eval` function that could interpret the language itself [18]. This was shown to be a very concise way to demonstrate the succinctness and power of Lisp in comparison with other "Blub" languages[1], and has been explained in detail by Paul Graham [8].

The shortcoming of this approach is that it relies on the concept of *meta-circularity*; the idea that a language can be interpreted in itself. While this is quite an elegant idea, introducing it to students too early on will do more to confuse than clarify the simplicity of Lisp, because in order to implement `eval`, one first needs to have a working `eval`. However, there are a number of other reasons students will benefit

[1]The term "Blub" was coined by Paul Graham to refer to a language which falls somewhere in the middle of the continuum of abstractness, such as Java or Python [9].

from learning a stripped-back, minimal Lisp early on.

1.1 Why teach Lisp?

Learning new programming languages is important not just so that students can add new tools to their toolkit, but so that they are challenged to alter the way they think about programming in general. Many students who first learn to code in a procedural language, such as Python or Java, approach programming in a particular way. For example, the use of recursion may seem unnatural to them, even at times when recursion would be more appropriate or even more efficient than iteration [26].

In fact, students are often uncomfortable with more functional styles of programming. For example, a recent study in competencies with lambdas in C++ shows that "students had difficulty completing tasks using lambdas, but far less so with iterators" [32]. Lambdas are incredibly difficult to avoid, particularly for developers who wish to work on user interfaces, so while some may suggest removing them from the curriculum because students find them confusing, we advocate that they should be given greater emphasis in order that students might become more comfortable with them.

The effect that language has on the way we think has long been recognised. W. V. Quine noted that "Conceptualization on any considerable scale is inseparable from language" [24]. This fact is not limited to natural languages, but has application in the realm of programming languages too.

As Bach is to Sibelius in music, Lisp is to common modern languages. Lisp is at one time both alien and fundamental to the languages students are often familiar with, so it has great potential for shifting students' problem solving paradigms. Lisp is unique for the way that it blurs traditional edges in language design. It encourages students to consider functions as objects through the use of first class procedures, and it allows programmers to treat data and code in the same way, with its unique macro system that allows meta-linguistic programming. Additionally, it provides ways for students to come to terms with functional composition, lambda expressions and other functional programming concepts without the overhead of type inference that a language such as Haskell would require a student to learn. For example, students who use Lisp will likely be much more comfortable with using more functional aspects of other languages, such as Python's `map` and `reduce` functions.

Another concrete example of the practical benefits of learning Lisp is the language Elixir—a functional language which runs on the Erlang Virtual Machine and is gaining popularity, particularly for Web Development [31]. Elixir includes many

benefits such as immutable data structures and types, high concurrency and support for Lisp-like macros. As languages like Elixir and Clojure gain popularity, the benefits of students having exposure to macros and functional programming concepts increases.

In the words of C.S. Lewis, "There are some things about your own village that you never know until you have been away from it" [15]. We must push students to leave the village of imperative programming paradigms if we wish them to fully understand both its benefits and its disadvantages.

1.2 Methodology

We have established our reasons for wanting to teach students Lisp. Unfortunately, often when students do learn a new language, they look first for constructs which they are comfortable with and revert to those. For example, when first shown Lisp, experience shows us that students who have learned Python may tend towards using for loops, even though these are unidiomatic to Lisp.

The consequences of one's first programming language were observed by Wexelblat. He noticed that "Programmers tend to favor their first language even to the extent of applying the style or structures of this language when programming in other languages" [34].

In light of this, we have sought to create a Lisp which removes some of these features completely, restricting students to use the language as it "should" be used.

This report describes the design and implementation of such a minimal Lisp. We have called this dialect of Lisp "Mary". Mary differs from McCarthy and Graham's Lisps as it does not contain a meta-circular `eval` function. All interpretation is done in the core—a minimal Lisp interpreter which we have written in Python. The core has an extremely limited set of built-in axioms, and all other functionality is bootstrapped on via a set of standard libraries.

We have used Python to write the interpreter, as this is a language that is commonly used for teaching first year University students, including the first year programming course taught at our university. We are not relying on any of Python's unique features for our implementation; Mary could easily be rewritten in any language.

We have also aimed to have Mary's minimality reflected by the source code of the interpreter, so that as an educational exercise, the student could be shown the source code and understand it, perhaps even rewriting a section of it. To an extent, the smaller we make both the language and the interpreter, the more freedom the

student will have to create and recreate parts of the language.

As well as a minimal core, some of the distinctive features that Mary offers include: macros; a traceable, lexically scoped environment; I/O; referential transparency; and some basic debugging tools.

In order to assess Mary, we critique and compare it to other dialects of Lisp. Mary's effectiveness in achieving our educational goals would be very difficult to measure quantitatively as the concepts that we desire to teach students are not easy to test, so our assessment is of a qualitative nature. In our evaluation, we measure the extent to which Mary is minimal, and the degree to which it emphasises the concepts we desire to use Mary to teach by comparing it to other dialects of Lisp.

1.3 Report Outline

This report begins by collating and commenting on past work done in the area of educational and minimal Lisps, particularly focussing on the Minimal Lisp described by Paul Graham [8]. We then explain what motivations we have for creating Mary, and enumerate our goals for the project. After this, we discuss various aspects of Mary's design. These are organised into the categories of language features, features which particularly assist in education, and more advanced and extended features. Finally, we evaluate Mary and compare it to other similar languages using a set of criteria based on our design goals. In our conclusion we summarise the contribution that Mary makes to the literature and the results of our evaluation.

Chapter 2

Background

There is good evidence that Lisp is a helpful language for teaching functional and recursive programming. For example, Maniccam has investigated the use of Lisp for teaching algorithms written in what is described as a purely functional and recursive style and discovered that using functional programming to show students concepts such as recursion has many benefits and helps to broaden students' programming skills [17]. There have also been a number of efforts to create minimal or educational dialects of Lisp for various reasons.

2.1 Paul Graham's minimal Lisp

As mentioned earlier, McCarthy and Graham have shown how with only seven primitive Lisp functions, one can write an `eval` function which can interpret Lisp code [8, 18].

When writing a Lisp interpreter, the evaluation can be described by a single universal `eval` function. This function takes a Lisp expression and an environment, and returns the result of the interpretation of the expression in the environment. The environment is represented as a list of two item lists, each representing a symbol and a value. If the Lisp expression is an atom, it is looked up in the environment, and if it is a list, each item of the list is evaluated recursively and the first item in the list is applied to all of the other items in the list. This eval function itself can be written in Lisp, provided seven axioms are available; `car`, `cdr`, `cons`, `cond`, `atom`, `eq` and `quote`[1], and is able to interpret itself [8]. Graham's Lisp is dynamically scoped,

[1] Graham also uses `defun`, but this is a special form rather than an axiom, and could be removed by replacing all calls to functions with the function definitions themselves.

Figure 2.1: A LISP Machine at the MIT Museum. One of perhaps 7,000 machines prior to 1988 built and optimised to run Lisp code efficiently. Photo courtesy of Jszigetvari.

and indeed must be because of the way environments and closures are implemented using the stack. We provide a demonstration of Graham's interpreter in Appendix B. A Python implementation has been created by Valle, which we used as a starting point for much of our work [33].

Graham's approach is a clever way to show the Turing completeness of a very compact language, but one of its drawbacks is that its meta-circularity would cause some confusion for students who are wanting to use the language for learning purposes. There may be uncertainty over which `eval` function is being called when code is interpreted, and in a sense it is true to say that both are being called simultaneously.

Another limitation that Graham's approach has is that these seven primitive operations, once defined by the Lisp `eval` function, cannot be freely redefined in Lisp itself.

For example, three functions that are basic to all Lisp dialects are `cons`, `car` and `cdr`. Lists are the most elementary data structure in Lisp, and these are the functions used respectively for constructing a list from a single atom and another list, accessing the first item in a list, and accessing the rest of the list. If these are not defined in the interpreter, they can actually be defined in multiple ways, one of which involves only the use of lambdas, as shown below.

$$\begin{aligned} \texttt{cons} &\leftarrow \lambda xy.\lambda f.f(xy) \\ \texttt{car} &\leftarrow \lambda c.c(\lambda xy.x) \\ \texttt{cdr} &\leftarrow \lambda c.c(\lambda xy.y) \end{aligned}$$

Because of the meta-circular nature of Graham's two `eval` functions, there is no way that we could use these redefinitions to reduce the set of axiomatic primitives in the dialect.

Mary seeks to have the underlying interpretive language simple and modular, so that it is easy to remove and add functions to the core, then redefine them in a standard library in Lisp itself. Having an `eval` function in Mary would make this process confusing and much more difficult.

2.2 Other Related Work

Greg Pfeil has created a language based on Kell calculus in which he removed lambda from the base set of functions entirely, and defined even basic numeric representations in libraries in order to keep the core language small [21].

One notable example of a Lisp with a similar design approach to Mary is LispKit, which is a minimal description of Lisp involving seventeen built-in functions and a "purely functional" philosophy [14]. While this project shares much in common with Mary, there are some points of difference:

1. LispKit does not have the capability to explore macros, which are arguably one of the most important and unique aspects of Lisp. Likewise, because of its aims to be functional, LispKit does not support function definitions except through the use of `lambda` and `let`.

2. LispKit aims for being "purely functional", which means that I/O is generally not supported. One exception to this is an implementation of Turtle Graphics in LispKit, which adds two graphics functions to the base set [35].

3. LispKit does not seek to find a set of axioms that are as minimal as possible, but includes a number of redundant axioms, such as the arithmetic operators `add`, `sub`, `mul`, `div` and `rem`.

Having said this, Mary has benefitted from the influence of LispKit, particularly in thinking about the implementation of types. The Java implementation of LispKit by Krejic and Luzanin was formative for much of the design of the object inheritance structure in Mary [14].

Chapter 3

Mary – Philosophy and Design

We have examined some of the motivations for creating Mary. In this chapter, we elaborate on these motivations and explain some of the design decisions made in light of our goals.

3.1 Educational philosophy

The educational framework we have followed is Piaget's philosophy of Constructivism. Constructivism claims that students learn best when they are able to explore a domain and make mistakes; "constructing" their knowledge of the field as they conduct open-ended investigations [3].

Mary seeks to apply the Constructivist theory of education to the domains of functional programming, language design and computational thinking based on Church's lambda calculus, which was one of the core influences in the design of Lisp [18]. It does this by removing most of the scaffolding that regular dialects of Lisp provide so that students can learn to "construct" a language themselves.

For this reason, Mary provides extremely minimal functionality at its core. Though this restricts students, it also gives them the freedom to build a language up on top of it and to explore how a small number of functions can be combined to do very complex tasks.

3.2 Goals of Mary

As mentioned earlier, the central goal for Mary is to demonstrate to students the simplicity of Lisp; that with only a few basic functions as a starting point, a much more comprehensive language can be constructed.

This means our architecture involves an interpreter that defines these functions, and by default reads a Lisp library which defines more complex functions before other code is run. These libraries can be easily removed or altered and redefined by students, and student assignments could be set with just a small subset of these functions if greater restrictions are desired. The core functions are also malleable, as a new set of core functions can be easily defined by writing a new class that provides more or less core functionality.

We have currently defined multiple sets of default axioms to demonstrate this, which can be chosen with an option flag when running Mary. These sets include: an extremely minimal set that forces list manipulation functions to all be defined in the standard library; a set that does not include `defun`, and thus forces `defun` to be defined as a macro; and a set that is identical to the previous set except that it does include `defun` in the core.

As we believe simplicity will greatly aid in teaching, the goal for Mary is to be simple in both its design and implementation. We have attempted to remove clutter from the interpreter, the language itself and the libraries of Lisp functions outside of the core.

Mary also aims to provide helpful information for debugging, and has multiple levels of feedback to this end. This feedback could include a history of macro-expansions, a history of evaluation, and a trace of the environments each expression was evaluated in.

Mary does not support reassignment to variables. Graham suggests using variable reassignment in Lisp as if it had a tax on its use [7]. In Mary we have gone one step further by removing it altogether. This provides Mary with referential transparency.

Chapter 4

Core language Elements

In this section we describe the core functionality of Mary, beginning by enumerating the axioms of the language. We use the word *axiom* here to refer to built-in functions. These are contrasted with both standard library functions, which are defined natively in the language itself, and special forms, which aren't used in the same way as functions that simply take a set of parameters and return a value, as the axioms do.

4.1 Axioms

The six axioms of our language are as follows:

`quote` Returns the argument without evaluating it.

`if` Our alternative to Graham's axiomatic `cond`. (if a b c) will return b if a evaluates to a truth value, otherwise c. This evaluation is lazy.

`car` Returns the first item of a list or the first character of a symbol.

`cdr` Returns all but the first item of a list, or all but the first character of a symbol.

`cons` Returns a list or symbol where the first element is the first argument and the remainder is the second argument. (**cons** 'a '(b c)) will return (a b c), and (**cons** 'a 'bc) will return abc.

`atom?` Returns the argument if the argument is an atom, otherwise the empty list.

We have also decided to support overloading for list functions on symbols, which allows us to do string manipulation without adding any extra axioms.

4.2 Special forms

The three forms `lambda`, `defun` and `defmacro` are not so much axiomatic functions as they are definitional elements of the language. Graham does not count `defun` as an axiom in his language, as it is useful for legibility but is not completely necessary. Also, as he uses an environment which is itself a Lisp expression, he is able to define `lambda` and `label` in terms of the other axioms. This forces him to scope his language dynamically, as the environment itself is coupled tightly with each functional call. We do include these special forms in Mary. However, we have a second version of the core that does not include `defun` as an axiom, but defines `defun` as a macro. This shows that we could cut the core down to include one less special form, but we believe that the definition of `defun` as a macro, which involves nested quasiquotes and splicing, would be confusing to students who are learning Lisp for the first time.

4.3 Types

We have decided to implement types in Mary in an Object Oriented style. We have done this by creating an abstract LispExpression class, which is inherited by each Lisp type. This makes typing, and therefore interpreting, much easier. It also allows us to associate appropriate information and methods with different Lisp types.

The six types we have chosen to operate with in Mary are as follows:

List A generic list expression, eg. (a b c).

Symbol A symbol or string. This can include any character that is not a read macro, a bracket or a space.

Number An integer or float. These are both implemented using the corresponding Python types, but are represented by a single class in the interpreter.

Function A built-in Lisp function which is defined internally as a Python function.

Lambda An anonymous function, which is applied by creating a closure and performing search and replace rules on the function's body.

Macro Very similar to a lambda, but the body is interpreted (macroexpanded) in a dynamic environment rather than a lexical one with unevaluated arguments, and the result is then interpreted again in the current environment.

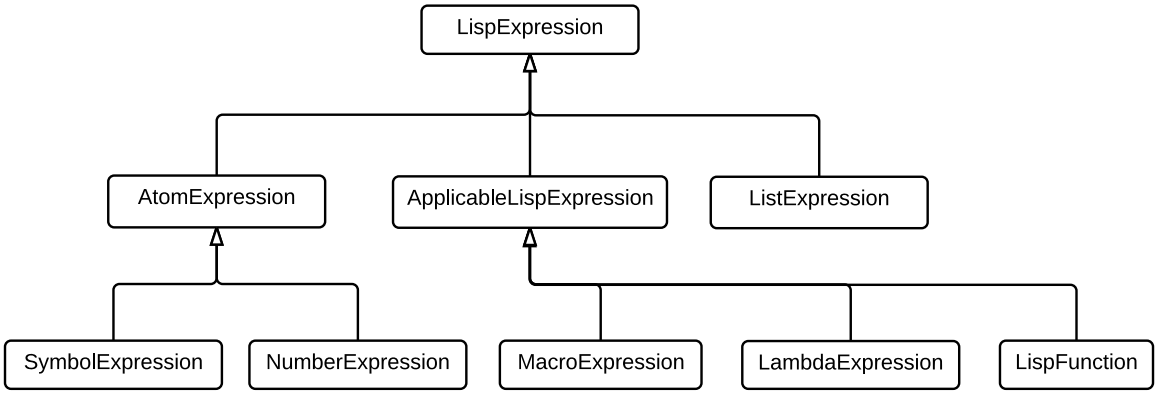

Figure 4.1: Diagram of inheritance for Lisp Expression types in Python

The structure, names and inheritance of these types as Python objects are shown in Figure 4.1.

We have chosen to implement booleans implicitly. Every expression is treated as having a "truthy"[1] value except for the empty list and the empty string, as is the approach of all major Lisp dialects. It may seem that one cannot have an empty string in Lisp when symbols and strings equate to the same thing, but we have defined `cdr` in such a way that the result of the expression (cdr 'a) is the empty string. By contrast, (cdr '(a)) evaluates to the empty list, which is equivalently false.

4.4 Arithmetic operations

In addition to these axioms, we have provided the two functions `subtract` (`-`) and `less than` (`<`), which offer the ability to do mathematical and string comparison operations. These operations are not necessary to make the language a functioning language. Without subtract operations, one could still define arithmetic operations in multiple ways, such as using Church numerals, but most of these methods would run against the grain of our minimalist design philosophy, and would produce more confusion for learners than helpful clarification.

The presence of the `less than` operator has allowed us to remove the `eq` function from McCarthy's original seven axioms, which returns true if and only if two atoms are equal. `eq` can now be defined as "(**not** (**or** (< a b) (< b a)))", and this definition can be extended recursively to work on lists as well as atoms. So although we might separate `less than` into a separate category from the core axioms, in reality it is

[1]That is, evaluating to true.

necessary for a lot of important computation.

As these are the only arithmetic operations provided in the core, and other operations such as `+`, `*` and `/` have been defined in terms of these two built-ins, arithmetic computation is generally inefficient in Mary. To improve efficiency, we could have given all arithmetic functions mappings to machine level instructions, but we chose to just use these two functions as our built-ins in order to keep Mary minimal, and show students how little was in fact necessary to create a comprehensive language.

4.5 I/O

We have created two functions, `printsym` and `inputchar`, which respectively print a single symbol to standard output and return a single character read from input. We have bootstrapped other I/O functions on top of these, including `readline` and `printline`, which deal with more characters or symbols. I/O is useful for educational and debugging purposes, but mostly just shows that our language can produce side-effects, despite its functional leanings.

Chapter 5

Educational Features

As we designed Mary to be useful in education, we have included a number of features that have educational uses.

5.1 Standard Library

We have created a set of standard libraries of Lisp functions defined in Mary, which our interpreter reads by default before doing anything else. Each set of axioms has a corresponding set of standard libraries to be read. For example, in the axiom set without `defun`, one of the libraries contains a definition of `defun` as a macro.

The lambda function is very powerful, and one of our goals with this standard library is to show students some of the power that anonymous functions can provide. It has been shown that lambda can be used for simulating iteration, imperative programming, variable assignment, and control transferral [29,30], and our standard library seeks to show some of the ways in which this is possible.

For example, we have included the following definition of `let`, adapted from Hoyte [12].

```
(defun firsts (l)
  ;; Takes a list of pairs, returns a list
  ;; of all the first values (cars) in each pair
  (if l
    (cons (caar l) (firsts (cdr l)))
  )
)

(defun seconds (l)
```

```
  ;; Takes a list of pairs, returns a list
  ;; of all the second values (cadrs) in each pair
  (if l
    (cons (cadar l) (seconds (cdr l)))
  )
)

(defmacro let (bindings &rest body)
  `((lambda ,(firsts bindings)
    (progn ,@body))
    ,@(seconds bindings))
)
```

We have also written a script to order functions based on interdependencies, and clearly label what other functions within a library a given standard library definition relies on. We have done this by topologically sorting the functions based on references by other functions which we obtained by examining the first item in any lists within the function definition. This means one can remove standard library functions and be aware of what other functions will be affected. Any functions that are dependent will be below the function being removed. By way of example, we provide this code snippet from the standard library.

```
...
;; pair?
;; DEPENDENCIES: NONE
;; DEPENDED ON BY: sort, max, reversestr, reduce,
;;   min, reverse, minlist, coerce, maxlist
(defun pair? (x) (cdr x))

;; reduce
;; DEPENDENCIES: pair?
;; DEPENDED ON BY: any?, all?
(defun reduce (fn l)
  (if (pair? l) (fn (car l) (reduce fn (cdr l)))
      (car l)))
...
```

If one removed `pair?` from the above standard library, one would also have to either remove or redefine `reduce`, which means other functions depending on `reduce` would also have to be removed. These functions would all be below `reduce` in the file, so one could simply cut the file off at that point and ask a student to solve problems using only functions above that point in the standard library.

We have also included a number of standard algorithms written in Mary, such as quicksort. As an example, here is our implementation of quicksort.

```
(defun quicksort (l)
   (if (cdr l)
       (append (quicksort (allwhich (cdr l)
           (lambda (x) (< x (car l)))))
       (cons (car l) (quicksort (allwhich (cdr l)
           (lambda (x) (not (< x (car l))))))))
  l))
```

In this function, `allwhich` is a function which takes a list and a function as parameters and returns the subset of the items in the list for which the function returns true. It is defined as follows.

```
(defun allwhich (l fn)
    (if l (if (fn (car l))
        (cons (car l) (allwhich (cdr l) fn))
        (allwhich (cdr l) fn))))
```

The function `append` is a generalised version of `cons`, which takes two lists and combines them into one. It is given with the following definition.

```
(defun append (lista listb)
    (if lista
     (cons (car lista) (append (cdr lista) listb))
     listb))
```

The function `not` is a predicate, and has the definition: `(if x () 't)`. All of the definitions used by quicksort are straightforward and only take a few lines of code. By seeing all these definitions, a student can witness the simplicity and elegance of quicksort as an algorithm. It is helpful to show students how few steps there are from a language with only a few axioms to an algorithm that sorts lists of items in $O(n \log n)$ time.

5.2 Environments

Mary has "lisp-1" style namespaces. This means there is a single namespace for both functions and variables, as opposed to lisp-2, which has a separate namespace for each. The advantage of this is that functions or macros that pass and return other functions or macros are much simpler to write [4]. Lisp-1 is used by Scheme, and other commonly used dialects of Lisp such as Clojure [23]. Hoyte argues that lisp-2 provides an advantage in that it "eliminates an entire dimension of unwanted variable capture problems" [12]. As variable capture is often an issue in working with macros, this is a consideration we made in the implementation of Mary. However

Hoyte also admits that "lisp-1 lisps do not suffer any theoretical barrier to macro creation", so we decided that lisp-1 would be the appropriate choice for Mary.

We desired to implement a lexically scoped environment rather than a dynamically scoped one, as we believe that lexical scoping will be much more accessible to students familiar with other common lexically scoped languages, such as Python or Java. This contrasts with Valle's Python implementation of Graham's interpreter, which is dynamically scoped, as new environments are created as expressions are interpreted, rather than when they are defined [33].

Bobrow has described an implementation of Lisp environments using a "Spaghetti Stack" structure [2]. This was improved upon by Steele who added some control structures to allow for static and dynamic scoping [28]. Our implementation was inspired by Steele, and can also achieve static scoping, although we have simplified Steele's multiple control structures into one Python class, called `Environment`.

The `Environment` class has a parent environment, and a set of bindings. These bindings are stored in a Python dictionary. When a lambda expression is read, it is stored as a lambda object, and assigned a parent environment. When this lambda is interpreted, it will create a closure inside the environment in which it was created, rather than the environment it was called.

This process occurs in the `apply_to` method in the `LambdaExpression` class, which we reproduce the Python code for below. The method `track_env` is used to maintain a record of the environment evaluation history to allow stack trace debugging, and `check_args` is used to ensure the right number of arguments were passed. The important lines are lines 17, 19 and 21. In these lines the new closure is created, the arguments are defined in the closure, then the body of the lambda is evaluated in this closure.

```
1  def apply_to(self, arguments, environment, caller, debug):
2      """
3      Apply to arguments in environment.
4      Eval info stored with caller for debugging if debug flag set.
5      """
6      # Arguments are evaluated first
7      arguments = ListExpression(
8              [argument.evaluate(
9                      environment, debug) for argument in arguments.value])
10     # Ensure the right number of arguments were passed
11     self.check_args(arguments)
12     # Copy the body to separate it during debugging if debug flag set.
13     # Keep track of the environment.
14     body = caller.track_result(self.body.copy(
15             )) if debug else caller.track_result(self.body)
16     # Create a closure to apply the lambda in
```

```
17        closure = body.track_env(self.environment.create_child())
18        # Define all arguments in the closure
19        self._define_args(arguments, closure)
20        # Evaluate the body in the environment
21        return body.evaluate(closure, debug)
```

When a `SymbolExpression` is evaluated, a lookup is made in the environment. If the binding is not present, the parent environment is consulted recursively until the default environment is reached.

This lookup is done by the environment itself, and can be seen in the following method from the Environment class.

```
def retrieve_definition(self, label):
    """Get definition of label in environment if it exists"""
    if not type(label) == SymbolExpression:
        raise TypeError("definition retrieval", label, "Symbol Expression")
    if label.value in self.definitions:
        return self.definitions[str(label)]
    elif self.parent_environment:
        return self.parent_environment.retrieve_definition(label)
    else:
        raise UnknownLabelError(label.value)
```

Due to its succinctness, students can read through this Python code and see how and when label lookups happen, and understand ideas such as scoping and variable capture in a deeper way.

5.3 Debugging and User Feedback

Upon request, Mary provides feedback on evaluation history at three levels. The three levels are macroexpansion tracing, standard debugging and environment debugging.

5.3.1 Macroexpansion Tracing

This feedback shows users all the macros which were expanded during interpetation of the program. For example, in the version of Mary with no built-in function definitions, `+` is defined as the following macro:

```
`((lambda ,'(x y) ,'(- x (- 0 y))) ,x ,y))
```

So whenever this macro is invoked with macroexpansion tracing, this definition is revealed, along with the result of the expansion. For example:

```
>(+ 1 2)
-(macro + (x y) `((lambda ,'(x y) ,'(- x (- 0 y))) ,x ,y))
=((lambda (x y) (- x (- 0 y))) 1 2)
3
```

This allows students to see macroexpansion in action, and to trace how macros are evaluated in order to understand macros and their power better. Mary's macroexpansion is explained in more detail in section 6.1.

5.3.2 Standard Debugging

Standard debugging allows the user to see the entire evaluation history of an expression. For example, in the following stack trace, we see the history of evaluation for the expression (+ 2 3).

```
> (+ 2 3)

Evaluation History:
->(+ 2 3)
--|->+
==|=>(lambda (x y) (- x (- 0 y)))
--|->2
--|->3
=>(- x (- 0 y))
--|->-
==|=>- [builtin]
--|->x
==|=>2
--|->(- 0 y)
--|--|->-
==|==|=>- [builtin]
--|--|->0
--|--|->y
==|==|=>3
==|=>-3
=>5
5
```

Expressions at the same depth level are either equivalent, or parts of the same list expression one depth higher.

Arrows with equal signs (=) represent evaluations of a previous expression, connected to their unevaluated expressions by a pipe (|) if they are more than one line away from them.

For example, in the above trace, the expression (– 0 y) evaluates to the number -3. The first item in the list is - (subtract), which evaluates to a built-in function, the second is 0, which is a number so needs no evaluation, and the third is y, which evaluates to the number 3. The result of (– 0 3) is -3, so these are connected by a pipe.

A full and formal explanation of stack tracing is provided in Appendix A.

5.3.3 Environment Debugging

Environment debugging allows the user to see not just the trace of evaluation, but the environments in which the bodies of lambda expressions are evaluated. Each environment is represented with a chain of name spaces, in which the lower spaces take priority, and the highest environment represents the default environment, including any built-ins and defined functions. With verbose debugging, the previous stack trace would look like this:

```
> (+ 2 3)

Evaluation History:
->(+ 2 3)
--|->+
==|=>(lambda (x y) (- x (- 0 y)))
--|->2
--|->3
=>(- x (- 0 y))
  {DEFAULT ENVIRONMENT}
           ||
  {'y': '3', 'x': '2'}
--|->-
==|=>- [builtin]
--|->x
==|=>2
--|->(- 0 y)
--|--|->-
==|==|=>- [builtin]
--|--|->0
```

```
--|--|->y
==|==|=>3
==|=>-3
=>5
5
```

At the point that the expression (– x (– 0 y)) needs to be evaluated, a new environment is created. This is shown in the stack trace. As nested environments are added, the student is able to see how and when closures are formed. This helps them to understand lexical scoping. In the lines after the environment's creation, the symbols `x` and `y` have the values `3` and `2`.

As an example which could be used in teaching, the expression:

```
(let ((x 3) (y 2))
        ((lambda (y)
                (- x y)) 1))
```

generates the following in its environmental stack trace:

```
...
==|==|=>(- x y)

  {DEFAULT ENVIRONMENT}
          ||
  {'x': '3', 'y': '2'}
          ||
      {'y': '1'}

--|--|--|->-
...
```

A student can see here that when the lambda expression is called, a new environment with a higher priority is created. In this environment, the value of `y` becomes `1` rather than `2`, which it was in the parent environment, but the value of `x` remains as `3`, as it is not overridden in the new closure.

One can demonstrate the effects of lexical scoping by adjusting the brackets in this expression, so that the anonymous function is defined, returned and then called. The expression:

```
((let ((x 3) (y 2))
        (lambda (y)
                (- x y))) 1)
```

will return the same result as the expression above, but will produce a different stack trace. Students can examine the differences to see how lexical scoping works.

5.4 Error messages

On top of these three levels of feedback, we have included a number of built-in Lisp errors. These are raised in response to syntactic or semantic mistakes in code. When these errors are raised, a stack trace is printed out to aid in debugging the problem. For example, if one attempted to add a number to a symbol, the response would be as follows:

```
> (+ 1 'a)

Evaluation History:
->(+ 1 'a)
--|->+
==|=>(lambda (x y) (- x (- 0 y)))
--|->1
--|->'a
--|--|->quote
==|==|=>quote [builtin]
--|--|->a
==|=>a
=>(- x (- 0 y))
! Bad type: - expected Numbers but got 0 and a
```

This error was raised by the built-in function `-`. Hopefully the stack trace and error message would provide enough information to show a student that the error had been introduced by the calling of an arithmetic function on the symbol "`a`".

A study on feedback for Lisp users showed that the preferred methods of display for novice or occasional users were range highlighting, interlaced displays of evaluation results and simultaneous displaying of called functions [11]. It would be helpful to introduce range highlighting and functionality to step through and see function calls to our results display. However, we believe that the interlacing of results provided by this simple text based interface is a good compromise between simplicity of implementation and usefulness of feedback.

Chapter 6

Advanced Features

The following features are of an advanced nature, and would be useful for teaching students some of the more important and unique capabilities of Lisp.

6.1 Macros and Quasiquotes

One of the limitations of Graham and McCarthy's descriptions of minimal Lisps is that neither has the ability to process Lisp macros [8, 18, 19]. Although macros are one of the most compelling and unique things about Lisp as a language [7, 12], they weren't introduced in to the language until the mid 1960's [27]. It may be largely for this reason that the use of macros in a minimal Lisp has not yet been explored in depth.

While Graham describes "seven primitive operators", he has decided not to count the keyword `defun` as an operator [8]. We believe this is fair, as he shows this keyword is unnecessary provided the environment is stored as a Lisp expression and the language is dynamically scoped. However, in desiring to add macros to Mary, we wanted to consider whether we could remove anything else from the core. We concluded that functions could be described as macros, so a function to create labelled functions could be described by a macro building macro. This meant we could remove `defun` from the core.

We have done this in one of our versions of Mary, but have left `defun` in the default version to help avoid confusion for students.

A macro is a piece of code that generates code that is then interpreted. Generally, after a Lisp parser has read in some Lisp code and before this code is interpreted, a stage called macroexpansion happens. A macroexpander will go through the code

and find any macro calls. It will then expand these macro calls according to the definition of the macro [7].

To make macros readable, we needed to implement quasiquotations. Quasiquotation is another feature that has not been included in other minimal versions of Lisp. This is partially because they were not added to the language until the 1980's, and partially because they don't provide any new functionality to a language, but are really just convenient syntactical sugar which mostly aid the reading and writing of macros [1]. Quasiquotes (`) are much like the read macro `quote` (`'`), which tells the interpreter not to evaluate its argument, except that they can be escaped via the use of a comma (`,`). For example, the following Lisp expression would evaluate as follows.

```
>`(hello ,(cadr '(Alice Bob Carol)))
(hello Bob)
```

Instead of using a comma to escape a quasiquote, one can also use a comma-at sign (`,@`) to "splice" the resulting list expression in to the list above. For example, the following would evaluate as given.

```
>`(hello ,@(cdr '(Gdodbye my friend)))
(hello my friend)
```

These quasiquotes do not provide new functionality, but they do make macro writing much easier [7]. We show with the following equivalent expressions that anything that can be described with quasiquotes can also be described without them.

`(f ,c g) ↔ (**list** 'f c 'g)

`(f ,c ,@some−list d) ↔ (**cons** 'f (**cons** c (**append** some−list 'd)))

With quasiquotes, we are able to use macros to define `cond` as follows:

```
(defmacro cond (&rest options)
    (if (car options)
    `(if ,(caar options)
        ,(cadar options)
        (cond ,@(cdr options))))
)
```

This will expand out to a nested `if` statement during macroexpansion, then arguments will be evaluated lazily once the expanded statement has been created.

To write `defun` as a macro, we needed to use nested quasiquotes. Bawden has described a number of common combinations of nested commas, quotes and comma-at symbols inside nested quasiquotations, and some of these proved rather useful [1].

Our definition of defun is as follows:

```
(defmacro defun (name params &rest body)
     `(defmacro ,name ,params
          `((lambda ,',params ,',@body)
          ,,@params)
      )
)
```

Many of the combinations of symbols here may seem mystical, however we have found the following descriptions by Bawden enlightening [1]:

`,'` The value of X will appear as a constant in the intermediate quasiquotation and will thus appear unchanged in the final result.

`,,@` The value of X will appear as a list of expressions in the intermediate quasiquotation. The individual values of those expressions will be substituted into the final result.

We add to this list the following:

`,',@` The value of X will appear as a list of constants in the intermediate quasiquotation. The individual values of those expressions will appear unchanged in the final result.

Although this definition is complex, we believe it is useful for students to see macroexpansions at work in order to better understand both Lisp and programming in general. Therefore it may be helpful to see complex programs described with macros rather than with functions. For this reason, we have not made this the default setting for Mary, instead providing it as an option for students wanting to learn more about macros and Lisp interpretation, and we have defined `quasiquote` as a core function.

As mentioned in section 5.3.1, we have included in Mary a macroexpansion tracing mode. This allows the user to see each macroexpansion as it happens, and follow the execution tree down to the smallest grain size. This means that students can see the two stages of macroexpansion and evaluation and their results.

One difficulty in using macros in this way is that functional recursion is no longer possible. If the macroexpander were to begin by expanding all the code out before the interpreter did the job of actually applying the code to the data, the macroexpander would continue expanding indefinitely. Graham says "Although the expansion function of a macro may be recursive, the expansion itself may not be" [7]. In other words, macro recursion can terminate if it is based on the structure of the data, but not if it is based on the data itself.

With most Lisp interpreters, this is certainly the case. For instance, MacLachlan's Python compiler for Common Lisp runs all macroexpansion a long time prior to any interpretive work is done [16]. This is done in order to improve efficiency, as this expanded code is then optimised into a representation of implicit continuations.

Because efficiency has not been a concern for us, we redefined how macroexpansions are executed in Mary. All macroexpansion in Mary is done by the interpreter, and is interlaced with interpretation itself. A macro is expanded at the point where it is found during interpretation, and then interpreted immediately. This means that it becomes possible to define macros which are recursive, and also means that functions defined using the above `defun` function are able to safely call themselves.

6.2 Gensyms

In order to avoid naming collisions in nested macro environments, we have provided a built-in gensym function. A gensym function is guaranteed to return a unique symbol in the environment it is called. Our implementation returns successively incremented versions of a global counter appended to a hash. Our parser will not accept a symbol beginning with a hash. Gensyms are a common and effective way of avoiding the problem of variable capture [12]. However, we have not needed to use this technique in any of our standard libraries because the language has lexical scoping and a unique dynamic macro evaluation system.

6.3 Multiple Dialects

We have alluded to the fact that Mary is not a single minimal dialect of Lisp, but rather a set of possible minimal dialects. There are three major dialects we have provided, but we have also given scope for the creation of new dialects with new sets of axioms.

The dialect we have introduced with no `defun` in its set of axioms is mostly equivalent to the dialect which does have `defun`. The one difference is that our definition of `defun` using a macro is incapable of creating functions which expect variable numbers of arguments.

To accept variable numbers of arguments, we have hard-coded into the interpreter the keyword `&rest`. If this keyword is included in a list of parameters to a macro or lambda, any remaining arguments will be coerced into a list which is given the label which succeeds `&rest` in the list of parameters. The problem comes when defining

`defun` with macros, as the `&rest` parameter is consumed by the first macro definition. Therefore it cannot be used in the lambda which represents the function body.

The following definition of `and`, which takes a variable number of parameters and returns an empty list if any of them evaluate to false, demonstrates the use of the `&rest` keyword:

```
(defmacro and (&rest vals)
    `(cond
        ((null? ',(cdr vals)) (if ,(car vals) 't))
        (,(car vals) (and ,@(cdr vals)))
    )
)
```

Other than this difference, the two dialects are identical in functionality, although execution tracing will yield different results. This comparison can be a useful educational tool, as students can be shown the difference between interpreting a statement using macros and without.

The third dialect is an extremely minimal version of Mary which only includes the built-ins `quote`, `if`, `lambda` and `defun`. This may seem extremely limiting, however, the following snippets from this standard library of this tiny dialect show that perhaps more can be achieved by this language than one might assume.

```
;; and
(defun and (x y) (if x y x))

;; or
(defun or (x y) (if x x y))

;; cons
;; DEPENDENCIES: NONE
;; DEPENDED ON BY: firsts, append, seconds, reverse, map
(defun cons (x y) (lambda (f) (f x y)))

;; car
;; DEPENDENCIES: NONE
;; DEPENDED ON BY: cdar, caadr, cddar, map,
;; reverse, append, cadr, caar, cadar, reduce, cond, cdadr, caddr
(defun car (c) (c (lambda (x y) x)))

;; cdr
;; DEPENDENCIES: NONE
;; DEPENDED ON BY: cddr, cdar, firsts,
;; seconds, caadr, cddar, map, pair?, reverse,
;; append, cadr, cadar, reduce, cond, cdadr, cdddr, caddr
(defun cdr (c) (c (lambda (x y) y)))
```

```
;; map
;; DEPENDENCIES: cons, cdr, car
;; DEPENDED ON BY: NONE
(defun map (fn l) (if l (cons (fn (car l)) (map fn (cdr l)))))

;; reduce
;; DEPENDENCIES: cdr, car
;; DEPENDED ON BY: NONE
(defun reduce
   (fn l) (if (cdr l) (fn (car l) (reduce fn (cdr l))) (car l)))
```

More exploration remains to be done into the possibilities that this dialect of Mary provides for learning about programming language capabilities and the simplicity of Lisp.

Chapter 7

Evaluation

Mary is a comprehensive language for its size. In order for students to see that Mary is not limited in functionality by its small size, the libraries include complex functions. Some of the standard library functions we have created include: selection sort; quicksort; a Fibonacci sequence generator; arithmetic multiplication, division and modulo operations; and higher order functions such as map and reduce. We have also implemented a game of tic-tac-toe in Mary to give a toy example of how Mary could be used as a "real" programming language.

7.1 Minimality

In total, Mary has 15 functions and special forms in its core, which is twice as many as Graham has [8], although these break down into categories of varying necessity. We define six core axioms, replacing Graham's `cond` with `if`, and `eq` with the arithmetic function `less than`. As well as the default core, there are cores with less axioms available for experimentation, and the option to roll your own default environments.

7.2 Correctness

We have written a series of 114 tests to run and confirm that the algorithms are behaving as expected for all of the standard library functions we have implemented in Mary. We use this to test the implementation of the language. As all tests pass, we are confident that we have produced a language which is robust.

As the language is functional and generally side effect free, we can also use formal

	MIT/GNU Scheme	Mary
Core Language	C	Python
Files in Core Language	532	8
LOC in Core Language	2896994	807
Files in Lisp	1	8
LOC in Lisp	1032	104
Total Number of Files	1661	13
Total Lines of Code	3229969	911

Table 7.1: Comparison between source code for Mary and source code for MIT/GNU Scheme

logic to prove the correctness of algorithms written in Mary. Assuming that the primitive axioms hold, one could write a formal program specification and translate it into Lisp code using Mary, then prove its correctness with a set of logical equivalences.

7.3 Comprehensibility

As one of our goals was to show students the simplicity of Lisp, we compare our interpreter with a common Scheme interpreter, MIT/GNU Scheme [10]. We ran `cloc` on the source code for both this interpreter and Mary to count statistics on the lines of code. Results are shown in Table 7.1. Prior to running this test we removed extraneous material from Mary, such as the code for the extremely minimal version and the tests we have written.

These results show that Mary is a much smaller interpreter, with a much greater proportion of the language written natively in Lisp (in Mary itself). As we sought to write as little as possible in Python, and to transfer the weight of building the language to the standard libraries written in Lisp, this is a positive result. 900 lines of code will be much less overwhelming for students than over three million.

We also compared Mary to an implementation of LispKit, which was created with similar aims. We found an implementation of LispKit written in Pascal and Visual Basic that had a total of 1021 lines of code, which is comparable to Mary [20]. However, LispKit does not include macros or I/O, and it includes more axioms than Mary. Furthermore, the source code is relatively difficult to understand, as it is written in languages that are unfamiliar to modern learners, and documentation is

scant. We believe that the source code of Mary is much easier to navigate.

7.4 Feedback to User

We believe that the feedback Mary generates presents a unique contribution to the world of minimal Lisps. In an effort to be minimal, some other Lisps, such as Graham's Lisp, do nothing to produce helpful error messages [8]. We have included both meaningful error messages and stack tracing. This means that students learning the language will not be as confused by error messages which are either unrelated to the actual fault, or are actually errors from the parent language and are unrelated to the Lisp code itself. The stack tracing that Mary provides includes environmental tracing, which allows students to see how closures are created during interpretation. We have also included a macroexpansion tracing mode, which allows students to see how macros are expanded and interpreted. We believe this will be an extremely educational tool.

7.5 Efficiency

Mary is for purposes of education rather than efficiency. Therefore, our goal has been to implement an interpreter that is really minimal and concise, rather than extensive and practical. Despite its lack of efficiency, Mary is useful for exploring topics of computational complexity with students. The overhead of creating closures and on the fly macro rewrites means the difference between quicksort and selection sort for example is tangibly large even for small lists. Mary's inefficiency is largely due to its lack of ability to do anything like random access on any structures. This is a limitation inherent to the linked list structure of Lisp dialects, which is usually overcome by having a more comprehensive interpreter. As everything in Mary is represented as a linked list, many common algorithms may not be as efficient in this language. We have also opted to not implement tail-call optimisation, which would aid in efficiency. Another factor which inhibits Mary's efficiency is the fact that it is written in Python. Because of this, we cannot expect Mary to outperform Python, which is already a slow language in comparison to other languages such as C and Java.

To measure the performance of Mary, we compared our implementation of quicksort in Mary with an implementation in Python. We chose to compare Mary to Python as this is the language we wrote Mary in. This comparison shows how much speed was lost from the parent language Mary was written in, and compares the

trends in execution speed. As well as the overhead of needing more interpretation, we expected loss in speed because of the lack of random indexing for any data structures. However, we also expected to find that as the size of the list being sorted increased, the increased time taken by Mary and Python would be proportional to one other, and that both would be $O(n \log n)$. The results of this measurement are shown in the graphs in Figure 7.1. We have included a regression line for the formula $a(n \log n) + b$ for both graphs to demonstrate their asymptotic complexity. To run these tests we had to disable debugging mode in Mary in order to avoid unnecessary copying of stack tracing which slows down execution speeds. The coefficient in the regression line for Mary is 1550 times that of Python, so this shows Mary is around 1550 times slower than Python for this task. The regression has a standard error of 1.38% for Mary and 0.96% for Python, so both algorithms are well within the bounds of $O(n \log n)$.

The implementation of quicksort in Mary involves an append operation, which means that each recursive call must do on average another $\frac{n}{2}$ operations, as there is no efficient way to append linked lists without tail pointers. Each call also does two separate calls on the list tail to find data which is smaller than the pivot, and data which is greater than or equal to the pivot. This means another n operations are done by each recursive call. As Python lists support random indexing, the Python implementation of quicksort is inherently more efficient, though not as elegant. While Python does around $n \log n$ operations, the Mary implementation does about $\frac{5}{2} n \log n$.

As an example of the inherent difference in efficiency between Lisp and languages with array structures available, compare the difference between an array implementation of a dequeue and a singly linked implementation. With an array, appending and removing to the right side is an $O(1)$ operation, while the same operations on the left hand side involve $O(n)$ time, due to the need to shuffle all other elements in the array along by one. In contrast, with a singly linked list, appending and removing from the head is always $O(1)$, while appending and removing from the tail takes $O(n)$ time.

If this was the only difference between the two structures it would seem that the singly linked lists of Lisp are no worse than arrays available in other languages. However, many common algorithms such as binary search and heapsort require random indexing, so cannot be implemented using a singly linked list. While there are advantages to linked lists over arrays, such as fast shuffling and dynamic memory allocation, most non-Lisp languages have capabilities for creating both linked lists and arrays. Array-like random indexing cannot be done in Mary (aside from somehow using closures as a data structure), so along with other dialects of Lisp, this is one

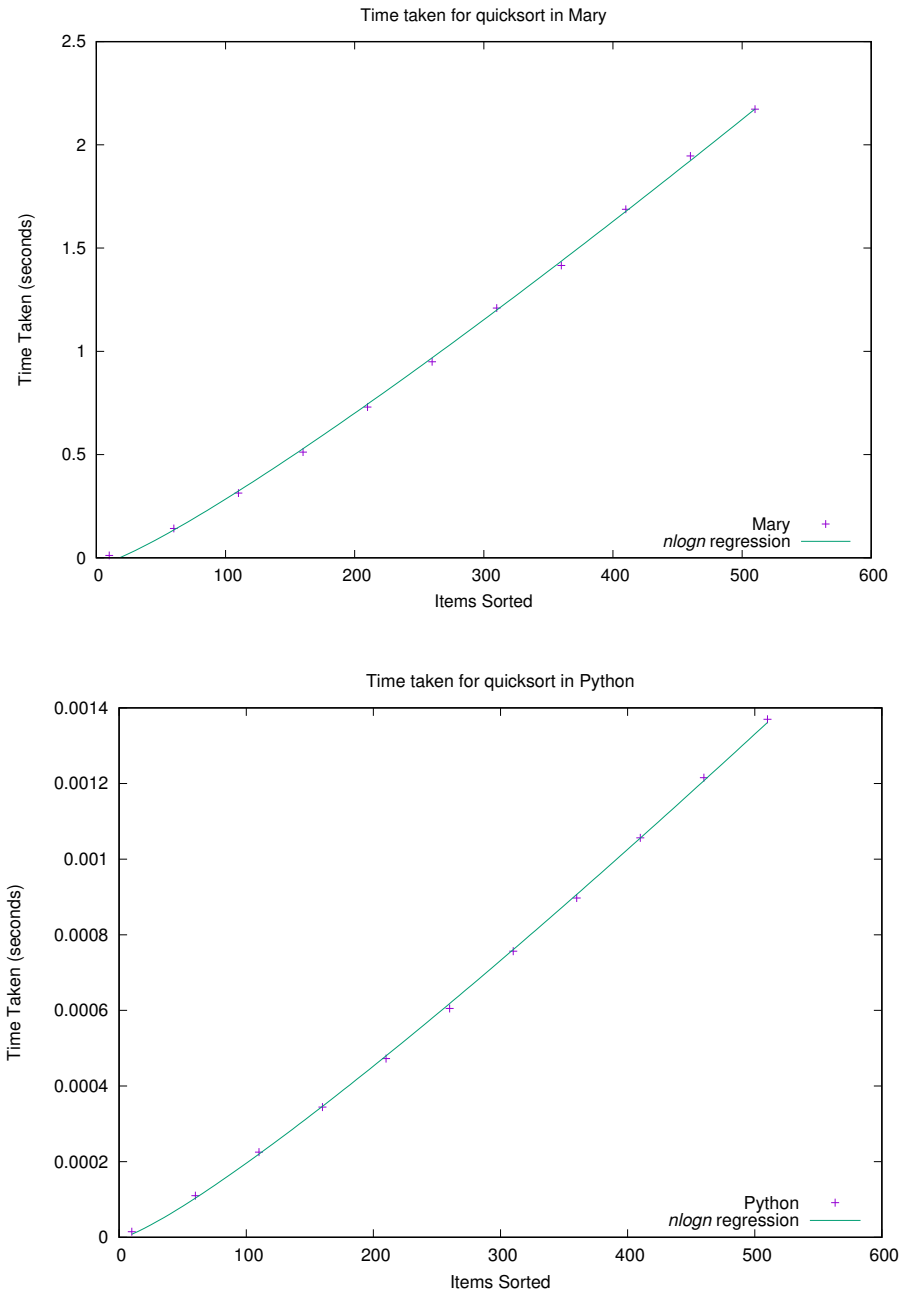

Figure 7.1: Performance of Quicksort in Mary and Python

of Mary's limitations. Other Lisp dialects have solved this by providing special data structures other than the list, such as vectors in Clojure. We have not provided any such structures in Mary, but rather suggest that this limitation is an opportunity to show students how Lisp requires a different style of thinking to what is necessary in other languages. We hope this allows students to adjust how they approach programming, and if it does, we will have achieved the goals we set for ourselves in creating Mary.

Chapter 8

Conclusion

In Mary, we have created a dialect of Lisp that is minimal in both its set of axioms and its source code. Mary is built on less than 15 axioms and has 911 lines of code.

We have met our goals of building on Graham's Lisp by creating a language with lexical scoping, macros and I/O while maintaining referential transparency [8]. We have provided a language that is modular, minimal and not meta-circular. These are features not present in Graham's Lisp but which we deemed as necessary for our purposes.

The design goal for the creation of Mary was to create a dialect of Lisp which emphasised the elements particular and unique to Lisp. This dialect would expose students to new ideas so that they might begin thinking differently about programming. Our evaluation shows that the two areas we identified as being important for this dialect, functional design and meta-linguistic programming, are emphasised in Mary. Mary's minimality means that much of Lisp which does not align with our design goals is not present, so will not distract students from the concepts which we have emphasised and do wish to teach students.

Mary would be a useful tool for students to learn about functional programming paradigms, meta-linguistic programming and language interpretation. The standard libraries allow teachers to set tasks with a subset of the language to force students to think outside of the box. The simplicity and modularity of Mary opens the door for a wide range of educational activities. Mary should be used educationally as an approachable gateway to functional programming and macros. The project is available at `http://andybelltree.github.io/Mary/`.

8.1 Future Work

One useful future addition to Mary would be a tutorial for beginners. Students who have not seen any Lisp prior to Mary may find it helpful to have the option of being walked through the semantics of Lisp and some of the features of Mary.

It could also be useful to investigate ways to further decouple code which is used in creating a debugging stack trace in Mary from interpreter code. Having to include stack tracing code amongst evaluation code means that the source code is slightly less perspicuous than would be ideal for students wanting to see how the interpreter works. This is a worthwhile trade-off for good debugging feedback, as we see helpful error reporting as vital for an educational language. It would be useful, however, if we could find a way to have this done well without having to interpolate code for evaluation and stack tracing as we have done.

An educational study remains to be done on whether Mary achieves learning outcomes. One way of doing this would be to expose students to Mary during a semester of study and compare the ways they use both Lisp and other languages before and after this exposure. Our hope would be that such a study would reveal a tendency for students who have spent time using Mary to use constructs such as recursion and lambdas with greater ease, and to work more comfortably with macros. We would also hope that students who have studied using Mary would have a greater understanding of language concepts such as scoping, referential transparency, functional programming and meta-linguistic programming.

Bibliography

[1] Alan Bawden et al. Quasiquotation in Lisp. In *PEPM*, pages 4–12, 1999.

[2] Daniel G. Bobrow and Ben Wegbreit. A model and stack implementation of multiple environments. *Commun. ACM*, 16(10):591–603, October 1973.

[3] Catherine Twomey Fosnot and Randall Stewart Perry. Constructivism: A psychological theory of learning. *Constructivism: Theory, perspectives, and practice*, pages 8–33, 1996.

[4] Richard P. Gabriel and Kent M. Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988.

[5] Patrick Gombert. Lispy elixir. Internet: https://8thlight.com/blog/patrick-gombert/2013/11/26/lispy-elixir.html [7/11/2016], Nov 2013.

[6] Paul Graham. Lisp quotes. Internet: http://www.paulgraham.com/quotes.html [10/10/2016].

[7] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[8] Paul Graham. The roots of Lisp. Internet: http://lib.store.yahoo.net/lib/paulgraham/jmc.ps [28/3/2016], January 2002.

[9] Paul Graham. Beating the averages. Internet: http://paulgraham.com/avg.html [10/8/2016], April 2003.

[10] Chris Hanson. MIT/GNU scheme. Internet: https://www.gnu.org/software/mit-scheme/ [8/8/2016], May 2016.

[11] J. M. Hary, L. A. Cohan, and M. J. Darnell. Users' preferences among different techniques for displaying the evaluation of Lisp functions in an interactive debugger. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, pages 45–50, New York, NY, USA, 1988. ACM.

[12] Doug Hoyte. *Let Over Lambda*. HCSW and Hoytech, 2008.

[13] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[14] N Krejic and Z Luzanin. An implementation of Lispkit Lisp in Java1. Internet: http://perun.pmf.uns.ac.rs/radovanovic/publications/2002-prim-lisp.pdf [30/9/2016], 2002.

[15] C. S. Lewis. *Studies in Words*. Cambridge University Press, 1960.

[16] Robert A. MacLachlan. The Python compiler for CMU Common Lisp. *LFP '92 Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 235–246, 1992.

[17] S. Maniccam. Sorting and searching using Lisp, functional programming, and recursion. *SIGCSE Bull.*, 41(4):53–56, January 2010.

[18] John McCarthy. History of Lisp. *ACM SIGPLAN Notices - Special issue: History of programming languages conference*, 13(8):217 – 223, August 1978.

[19] John McCarthy. A micro-manual for Lisp - not the whole truth. *SIGPLAN Not.*, 13(8):215–216, August 1978.

[20] Paul McJones. Original OUCL PRG LispKit. Internet: http://www.softwarepreservation.org/projects/LISP/lispkit/LispKit.tar.gz/view [8/8/2016], May 2015.

[21] Greg Pfeil. Kilns: A Lisp without lambda. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, ILC '14, page 9, New York, NY, USA, 2014. ACM.

[22] Kent M. Pitman. Common Lisp: The untold story. In *Celebrating the 50th Anniversary of Lisp*, LISP50, pages 6:1–6:12, New York, NY, USA, 2008. ACM.

[23] Christian Queinnec. *Lisp in small pieces*. Cambridge University Press, 2003.

[24] Willard Van Orman Quine. *Word and object*. MIT press, 1960.

[25] Eric S Raymond. How to become a hacker. *Database and Network Journal*, 33(2):8–9, 2003.

[26] Emmanuel Saint-James. Recursion is more efficient than iteration. *LFP '84 Proceedings of the 1984 ACM symposium on LISP and functional programming*, pages 228–234, 1984.

[27] Guy L. Steele and Richard P. Gabriel. The evolution of lisp. In *History of programming languages—II*, pages 233–330, New York, NY, USA, 1996. ACM.

[28] Guy Lewis Steele, Jr. Macaroni is better than spaghetti. *SIGPLAN Not.*, 12(8):60–66, August 1977.

[29] Guy Lewis Steele Jr. Lambda: The ultimate declarative. Technical report, DTIC Document, 1976.

[30] Gerald Jay Sussman and Guy Lewis Steele Jr. Lambda: The ultimate imperative. 2015.

[31] Lau Taarnskov. Elixir – the next big language for the web. Internet: http://www.creativedeletion.com/2015/04/19/elixir_next_language.html [7/11/2016], April 2015.

[32] Phillip Merlin Uesbeck, Andreas Stefik, Stefan Hanenberg, Jan Pedersen, and Patrick Daleiden. An empirical study on the impact of C++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 760–771, New York, NY, USA, 2016. ACM.

[33] Kjetil Valle. Implementing (the original) Lisp in python. Internet: http://kjetilvalle.com/posts/original-lisp.html [8/3/2016], November 2013.

[34] Richard L. Wexelblat. The consequences of one's first programming language. In *Proceedings of the 3rd ACM SIGSMALL Symposium and the First SIGPC Symposium on Small Systems*, SIGSMALL '80, pages 52–55, New York, NY, USA, 1980. ACM.

[35] Putnik Zoran, Budimac Zoran, and Ivanovic Mirjana. Turtle walk through functional language. *SIGPLAN Not.*, 26(2):75–82, January 1991.

Appendix A

Semantics of Stack Tracing

We provide the following stack trace to demonstrate the symbolic semantics of our stack tracing.

```
0  ->(cadr '(a b c))
1  --|->cadr
2  ==|=>(lambda (l) (car (cdr l)))
3  --|->'(a b c)
4  --|--|->quote
5  ==|==|=>quote [builtin]
6  --|--|->(a b c)
7  ==|=>(a b c)
8  =>(car (cdr l))
9      {DEFAULT ENVIRONMENT}
10             ||
11      {'l': '(a b c)'}
12 --|->car
13 ==|=>car [builtin]
14 --|->(cdr l)
15 --|--|->cdr
16 ==|==|=>cdr [builtin]
17 --|--|->l
18 ==|==|=>(a b c)
19 ==|=>(b c)
20 =>b
```

At the lowest level (on the farthest left), we see a pipe connecting the expression

at line 0, to the expression at line 8, and then on to the result at line 20. The first expression is the initial unevaluated expression, the second is a translation of the expression from the lambda definition in line 2, and the final expression is the result of the initial expression being evaluated. As the second and third expressions represent the result of some evaluation, they are indented with equal signs (`=`). The first expression is unevaluated, so is indented with a dash (`-`).

Lines 1–3 and line 7 have one level of indentation, which means they correspond to parts of the expression at line 0. Line 1 is the name of the function in line 0, and line 2 is its definition. This fact is again represented by the equal signs at line 2, showing it is an evaluation of a previous expression (in this case, the expression at line 1). Line 3 is the argument in the expression at line 0. Line 7 represents the evaluation of line 3, with intermediate steps being shown one level deeper between line 4 and 6.

Once the expression has been translated, a new environment is created and shown from lines 9–11, in which `l` corresponds to the list `'(a b c)`. The expression at line 8 is interpreted in this environment. At line 13, the symbol `car` from line 12 evaluates to the built-in function `car`, and at line 18 the symbol `l` at line 17 evaluates to the list stored in the closure that was created at line 9.

At line 19 we see the result of line 14, then at line 20 we see the result of the whole evaluation, that is, the expression at line 0 and line 8.

Appendix B

Graham's Lisp

The following Lisp was defined by McCarthy in his 1960 paper, and translated into Common Lisp by Paul Graham [8, 18]. It is dynamically scoped, and assumes a language with only `quote`, `atom`, `eq`, `cons`, `car`, `cdr`, `cond` and will interpret a similar language. This code would be interpreted by a parent language, then be run on other code, so that two Lisp interpreters were running simultaneously. This is the meta-circularity which Mary seeks to avoid.

```
(defun list. (x y)
  (cons x (cons y '())))

(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list. (car x) (car y))
               (pair. (cdr x) (cdr y))))))

(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))

(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
```

```
   ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
   ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
   ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                (eval. (caddr e) a)))
   ((eq (car e) 'cond)  (evcon. (cdr e) a))
   ('t (eval. (cons (assoc. (car e) a) (cdr e))
              a))))
((eq (caar e) 'label)
 (eval. (cons (caddar e) (cdr e))
        (cons (list. (cadar e) (car e)) a)))
((eq (caar e) 'lambda)
 (eval. (caddar e)
        (append. (pair. (cadar e) (evlis. (cdr e) a))
                 a)))))
```