The Rabbit Programming Language: Functional Programming for a Scripting World

Evan Hubinger

Abstract

Functional programming offers tremendous benefits to the programmer—but most existing functional programming languages are difficult to learn and write code for. The Rabbit programming language is presented as a solution, built to be easy to learn and write while also capturing the full power of functional programming. The culmination of nearly three years of work by the author, Rabbit is a modern, functional, interpreted, dynamically-typed, open-source, Turing-complete scripting language built on top of Python. The entire Rabbit interpreter is written in pure, version-agnostic, dependency-free Python, able to be run out of the box with nothing extra to install on any Python interpreter. It is hoped that Rabbit will be able to bring functional programming into the new realm of scripting programming languages.

I. Rationale

Despite intensive study and research efforts devoted to it, functional programming has remained a highly esoteric endeavor. Surveys of programming language popularity consistently rank most functional programming languages near the bottom.[1] A recent informal study on trends within discussions of programming languages found that the functional programming languages studied—Haskell and Scala—were mentioned far more often than they were used.[2] These results showcase a fundamental disjoint between the professional literature—in which much recent research and study has been put into functional programming[3]—and common usage—in which functional programming is relegated to the sidelines. This paper intends to explain why it is important that functional programing be promoted, identify possible causes for its underutilization, and present the solution—a new functional programming language built for the average, everyday programmer.

Functional programming is a particular programming paradigm based on functions. In a functional programming language, functions are first-class objects, with the ability to be passed as arguments to other functions and returned by other functions. Those functions that take other functions—called higher-order functions—are the functional programmer's main tools, allowing

---

[1] A wide variety of different programming language ranking sites and services agree on this result. A small sample of rankings that prove this paper's point are provided: http://langpop.com/; http://spectrum.ieee.org/computing/software/top-10-programming-languages; http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html; https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language
[2] The site mentioned is http://www.i-programmer.info/news/98-languages/7684-haskell-most-talked-about-language.html
[3] The number of articles turned up by a quick search for all recent scholarly articles on functional programming (181,000 from 2010-2014) should indicate the truth of this statement, but for a more salient example, see this academic journal, dedicated entirely to nothing but functional programming: http://journals.cambridge.org/action/displayJournal?jid=JFP

the abstraction of operations described in terms of functions. Classic examples of higher-order functions involve a function that applies another function to every argument in a list, or a function that uses another function to determine whether an element of a list should remain or be removed. These two higher-order functions are respectively referred to as map and filter, and are built-in functions in most every functional programming language. Higher-order functions like these provide the functional programmer with powerful abstractions. Functional programming is not just about how it uses functions, however. Functional programming also takes different approaches to data and state. In pure functional programming, all data, all state, and all names are immutable. Among other things, this means functions can never have side-effects—they can never do anything that would affect the state of the program. This implies that a function called with the same parameters (in the same scope) will always return the same value. This property, known as referential transparency, allows programs to be much more easily reasoned about, because the programmer always knows that all the information that could possibly determine the result of a function call resides in the function's arguments. In any actual practical scenario, however, this is impossible, because any program will inevitably have to interact with the outside world—do input/output—and when that happens, side effects are inevitable. Different functional programming languages take different approaches to solving this problem. Haskell, for example, confines all side-effects to a special "main" function. Functional programming languages also typically don't support imperative "do this, then that" code style. This is because, in a purely functional programming language, functions only do one thing—return a value.

The given explanation of functional programming says a lot about what features functional programming lacks, but little about those that it adds. As John Hughes explains in "Why Functional Programming Matters:"

> Such a catalogue of "advantages" is all very well, but one must not be surprised if outsiders don't take it too seriously. It says a lot about what functional programming isn't (it has no assignment, no side effects, no flow of control) but not much about what it is. The functional programmer sounds rather like a mediæval monk, denying himself the pleasures of life in the hope that it will make him virtuous. To those more interested in material benefits, these "advantages" are totally unconvincing.[4]

As Hughes notes, the misconception that functional programming is all about removing useful tools is prevalent among "outsiders." It is likely that this misconception is at least partly responsible for the previously mentioned lack of functional programming language use. This paper intends to correct this misconception by demonstrating the material benefits of functional programming. To do this, four major material benefits of functional programming are identified:

1. Shorter, more concise programs.
2. Increased program modularity.
3. Ease of debugging program problems.
4. The opportunity for the use of powerful optimizations.

The first benefit of functional programming is the ability to write shorter and more concise programs. Functional programming enables this by allowing for additional types of abstractions not possible—or at least not commonly used—in imperative languages. One functional programming feature that contributes to increasing possible abstractions is the ability to create—and the emphasis on—higher-order functions. Higher-order functions enable complex operations—for example, walking two lists simultaneously and applying a function of two arguments to the elements at each position in the two lists to generate a new list, an operation commonly referred to as zipwith—to be abstracted to a simple function call. These abstractions

[4] Hughes's paper can be accessed at: http://www.cs.utexas.edu/~shmat/courses/cs345/whyfp.pdf

enable shorter programs both by reducing the need to repeatedly specify the same procedure, and by enabling the functional programmer to think about problems in terms of the general principle being applied, write that general principle up as a higher-order function, and reuse it. Additionally, since many of the operations that would be represented using special syntactic constructs in imperative languages—such as loops—are represented as functions, they can much more easily be reused. If one, for example, wants to reuse the body of a previous while loop—since in functional programming while loops are represented using tail recursion, where a function calls itself at the bottom of the function—that body is already available to the programmer as a function ready to use.

The second benefit of functional programming is the ability to write more modular code. Hughes lays out the benefits of modular programming:

> Modular design brings with it great productivity improvements. First of all, small modules can be coded quickly and easily. Second, general-purpose modules can be reused, leading to faster development of subsequent programs. Third, the modules of a program can be tested independently, helping to reduce the time spent debugging. […] It is now generally accepted that modular design is the key to successful programming […].[5]

Functional programming enables increased modularity by providing the programmer with additional tools with which to break up problems into smaller, more easily solvable sub-problems—the essence of what modularity, breaking things up into smaller pieces, entails. Functional programming accomplishes this by increasing the ways in which sub-solutions can be combined together to solve the original problem, thereby allowing the programmer to break up the problem in new ways, knowing that functional programming gives them the ability to

---

[5] Hughes op. cit.

"glue"—as Hughes refers to it—their sub-solutions back together.[6] The primary way in which functional programming does this is through the use of higher-order functions, which allow problems to be broken down theoretically in terms of the general principle being applied, represented as a higher-order function. Additionally, the representation of more constructs as functions in functional programming languages enables increased reusability of those functions, and hence increased modularity.

The third benefit of functional programming is increased ease of debugging. Two features of functional programming enable this: referential transparency and data immutability. Referential transparency, by ensuring that functions have no side effects, allows function return values to be mentally substituted for the function call, alleviating the burden of having to think about the effects of a function call, and when those effects will occur. Instead, the functional programmer need think only about the return value. Combined with the ability to treat all calls to the same function exactly the same, since there's no state to change the way the function works, this makes thinking about programs much easier, and finding bugs in them easier still, since the only thing that needs to be considered to debug a function is its return value.

The fourth benefit of functional programming is the opportunity for the use of powerful optimizations. Not only does functional programming allow the programmer to reason more about their program, it also allows the language to do the same. For example, referentially transparent functions can always be memorized—meaning that, because they always return the same value for the same arguments, if one is called multiple times with the same arguments, the previously computed value can be used instead of recomputing the value anew. Additionally,

[6] Hughes op. cit.

functional programming makes parallel computing much easier, because the functional programmer need not worry about managing shared state or timing side-effects, because there is no shared state and there are no side-effects, and since this is always true, all operations in pure functional programming can be automatically parallelized. Finally, although this optimization is possible in non-functional languages as well, the elimination of tail recursion from the stack is found in almost every functional programming language because tail recursion is so heavily used in them that making it faster is very important.

The benefits of functional programming having been established, this paper returns to the problem of the low usage of functional programming languages. While the goal of this paper is not the popularization of functional programming, it is hoped that by analyzing why functional programming languages are not more widely used now, a better functional programming language can be created that responds to the existing criticism. Three possible reasons for programmers not using functional programming are identified:

1. Functional programming is too difficult, or, functional programs take too long to write.
2. Functional programming is too different, or, functional programming is too hard to learn.
3. Functional programming is too specialized, or, functional programming isn't applicable to everyday problems.

The first hindrance to today's functional programming languages is that functional code is simply too hard to write. To some extent, that is true about functional programming in general. While a functional program is usually much shorter than its imperative—non-functional—counterpart, it comes at the expense of having to spend more time thinking about the problem before writing code to solve it. This is because the corner-piece of functional programming is breaking up a problem into sub-problems that can be described using higher-order functions, a process that takes thinking on the part of the programmer. With experience, however, functional programmers get better at thinking about problems in a functional manner, and this problem tends to abate. Additionally, shorter, simpler, and more concise programs—as enabled by functional programming—actually save the programmer more time in the long run, because of the ease of debugging much more concise, much more modular code—and less of it. This is particularly important because programmers commonly spend more time improving and debugging old code than writing new code. Thus, functional programming in general does end up being easier to write. But many functional programming languages work against ease of writing by requiring programmers to obey strict type restrictions, a concept referred to as static typing. While static typing can be very useful for performance reasons and for catching a variety of errors at compile time, it hampers the ease and speed at which code can be written and forces the programmer to abide by oftentimes needless restrictions.

The second hindrance to today's functional programming languages is that they are just too difficult to learn. In a discussion the author had with Princeton computer science professor Brian Kernighan regarding this project,[7] he remarked that, when he first tried to learn Haskell, he gave himself the problem of carrying out some simple formatting on a text document. After working at the task for a great deal of time, he could not figure out how it could be done in the language, and moved on to other things. If even a Princeton computer science professor struggles

[7] I met Kernighan while visiting Princeton in August of 2014 and had a chance to talk to him about the Rabbit programming language. In addition to being a Princeton computer science professor, Kernighan is also a co-author of *The C Programming Language*, and the K in K&R C.

with learning functional programming, it seems likely that either it is not being taught well, or the languages that exist are too complicated. Given the proliferation of free, online learning materials for functional programming—specifically for Haskell, the language that professor Kernighan was attempting to learn[8]—the latter is more likely. The very layout of one commonly used introductory guide for Haskell suggests a problem: what Kernighan wanted to do, input/output, and what is easily doable in nearly every modern programming language, requires special semantics in Haskell and isn't introduced until more than halfway through the book.[9] Haskell and most other functional programming languages place significant barriers in the way of programmers attempting to learn them—barriers such as special semantics for input/output. Recently, the field of functional programming languages has been dominated by these types of statically typed, compiled languages. While these types of languages can be useful—when computational speed is a priority, for example—they tend to take longer to write code for, because of the extra attention to types needed on account of static typing, and be harder to debug, on account of the fact that the programmer can't just boot up an interpreter to test out features. This main body of functional programming languages also tends to separate side-effects out in a way that makes the use of them entirely different both from the way functions are normally called in the rest of the code and the way that functions with side-effects are usually called in imperative languages. While separation of side-effects is important, it is also important that when side-effects are needed, they can be used in a way that is comfortable for the programmer. This focus on purely functional side-effect handling hurts these languages' ease of learning. Neil

[8] Many different Haskell learning materials exist. A sample of the most popular to prove this paper's point are provided: http://learnyouahaskell.com/; http://book.realworldhaskell.org/; http://www.haskell.org/tutorial/; http://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf
[9] The chapters of the book referred to can be accessed at: http://learnyouahaskell.com/chapters

McAllister in “Why New Programming Languages Succeed—or Fail” remarks that “Still, pure functional languages remain unpopular. Developers like innovative ideas, but they don’t like being forced out of their comfort zones.”[10]

The third hindrance to today’s functional programming languages is that they are too specialized for specific tasks. This was the criticism Kernighan presented explicitly later in discussion with him, and it is a criticism that, given the focus of most functional programming languages on mathematical tasks, including attempting to make the languages themselves mathematically tractable, is widely true of most functional programming languages. On the other side of the chasm, languages that don’t use static typing, are interpreted, and are built for everyday, common usage are usually referred to as scripting languages. By not having a scripting language,[11] the functional programming languages of today—by using special constructs for input/output, being statically typed, requiring compilation, and generally being too dissimilar from conventional imperative languages, thus pushing developers out of their “comfort zones”—serve to reduce the ease at which they can be learned, written, and debugged, thus alienating

[10] McAllister’s article can be found at: http://www.infoworld.com/article/2610284/application-development/why-new-programming-languages-succeed----or-fail.html
[11] While the truth of this statement depends on one’s definition of “functional language” and “scripting language,” it is near certain that nothing like Rabbit—a truly functional scripting language as specified in this paper—has ever been created before. To demonstrate this assertion, we will quickly deal with all of the programming languages proposed as possible functional scripting languages in the stackoverflow thread on the subject (http://stackoverflow.com/questions/733222/are-there-any-decent-scripting-languages-that-use-functional-programming). Lua, Python, JavaScript, and Perl are all true imperative scripting languages, forgoing many of the important parts of a functional programming language such as immutable data, prevention of reassignment, and lack of strict side-effect control. Haskell is a pure functional language, as was previously mentioned, and lacks important scripting features. Scheme (Guile being a subset), supports variable reassignment and doesn’t have strict controls on data immutability or side-effects, and being a Lisp, lacks the ease of learning required for solving the problem of this paper. Scala is compiled, and also not fully functional in controlling variable reassignment, data mutation, and side-effects.

themselves from a wide variety of potential users—a result evidenced by the current low popularity of functional programming. Thus, if functional programming is thought to be important, the need for a new functional programming language that is more similar to commonly-used imperative scripting languages, and thus easier to learn, while still providing for the power of functional programming, is demonstrated. In particular, given the proven utility of Python as a good language for beginners,[12] a language utilizing Python would be preferred. This paper presents the new Rabbit[13] programming language as the solution to these problems.

II. Specification

This section will be devoted to a technical specification of this paper's solution to the problem of providing a functional scripting language: the Rabbit programming language.[14] Much of this section will be borrowed from the simultaneously-released full Technical Whitepaper, but, due to length constraints, the entire contents of the Technical Whitepaper cannot be reproduced here. Instead, it is highly recommended that the reader read the entirety of the full Technical Whitepaper if a complete specification of Rabbit's semantics is desired.[15]

[12] Many different studies have found this result to be true. A sample of such studies is provided to prove this paper's point: http://link.springer.com/chapter/10.1007/978-3-540-25944-2_157; http://pdf.aminer.org/000/278/636/a_comparison_of_c_matlab_and_python_as_teaching_languages.pdf; http://dl.acm.org/citation.cfm?id=1151880; http://micsymposium.org/mics_2005/papers/paper20.pdf; http://tech.canterburyschool.org/pycon/teaching_pygame.pdf; https://www.python.org/doc/essays/cp4e/

[13] The name was chosen in honor of the Rabbit of Caerbannog, the killer Rabbit in *Monty Python and the Holy Grail*, since Rabbit is built on Python and Python is named after Monty Python (source: https://www.python.org/doc/essays/foreword/). Additionally, Pythons eat Rabbits.

[14] All of the code for the Rabbit interpreter can be accessed at: https://github.com/evhub/rabbit

[15] The full Technical Whitepaper can be accessed at: https://github.com/evhub/rabbit/blob/master/docs/Technical%20Whitepaper.md

The culmination of nearly three years of work by the author,[16] Rabbit is a functional, interpreted, dynamically-typed, open-source, Turing-complete scripting language built on top of Python. Rabbit is built on Python because of Python's established nature as a scripting language; by using the same libraries and allowing for interoperability with Python code, the goal is that Python will provide Rabbit with an existing scripting framework that it can neatly fit into. The entire Rabbit interpreter is written in pure, version-agnostic, dependency-free Python, able to be run out of the box with nothing extra to install on any Python interpreter. As of this writing, the Rabbit interpreter has been tested and found to be working with no modifications on Python 2.6.9, 2.7.8, 3.2.5, 3.3.5, and 3.4.1. While all tests were performed using CPython, Rabbit should work just as well on Jython, IronPython, PyPy, Cython, or any other full-featured Python interpreter or compiler.

Rabbit is, at its core, a functional programming language. That means all the built-in Rabbit data types are immutable, variable reassignment raises an error, all functions are first-class objects, and higher-order functions are both provided and encouraged. But Rabbit is also a scripting language. That means that Rabbit supports dynamically typed operations and function arguments and cleanly integrated input/output. Rabbit is also a very extensible language, both due to the meta-programming tools it provides, and the ability to write Python code to change and control it. This combination, together with the ability to make use of Python's extensive existing libraries, makes Rabbit a very useful language for any of the everyday programming tasks that Python is useful for. Rabbit's use of the Python standard libraries also gives it more

[16] Over 18,000 lines of code and three years of work went into the Rabbit programming language, every single line of it written by me. The first two years were spent mostly toying around with ideas and writing the basic infrastructure, and the third was spent in intensive coding to get everything finished.

freedom to be different with its syntax without putting too great of a learning burden on the new programmer who knows at least some of the Python standard libraries.

Rabbit's basic syntax is intended to be easy to write, and, if the construct exists in mathematics, similar to the equivalent mathematical notation. A Rabbit source file, always encoded in UTF-8, is separated into a series of commands by line breaks. Comments are denoted by a pound character and continue until the end of the literal line, as in Python. No special syntax is reserved only for the top-level of a command. All Rabbit's syntactical constructs are allowed in all parts of the code. Each command simply serves as a thing that will be evaluated that can contain syntactical constructs. If, as is very commonly the case, the programmer wants to span one command over multiple lines, Rabbit will connect any lines that begin with whitespace to those preceding them. The whitespace is kept, however, and is used by certain constructs—such as code blocks or statements—to determine where things begin and end, and the indentation used to determine what is subservient to what. Other than for indentation, string literals, and in-between words (which can either mean a variable name with space in it or an automatic multiplication or function call), whitespace is ignored.

Variable assignment in Rabbit is much like in other languages, with the change that the evaluation of the value of the variable is deferred until later by default. This is done both as an optimization and to allow functions like Meta.unused to exist that can tell the programmer whether or not a variable has ever been used. This form is lazy assignment, where a variable is bound to a piece of code that will be evaluated when the value of the variable is needed, and is done via the equals operator. Direct assignment, where a variable is bound to the result of evaluating a piece of code, is done via the colon-equals operator. Neither operator, however, will allow a variable to be redefined unless it is being redefined to exactly its current value, or a special impure function, def, is used. Rabbit also supports assignment to lists of variable names, which will break up the result and attempt to assign its parts to each part of the variable name list, grouping extra items in the last variable.

Rabbit's functional features are built right into its basic syntax. When first evaluating a file or input from a console, side-effect producing functions are available for the programmer to call just like any other function. All functions are call-by-value and all Rabbit code is evaluated linearly from top to bottom, in sequence, to make imperative-style calling of side-effect producing functions easy. To simultaneously also allow for all of the benefits of referentially transparent code, Rabbit provides the programmer with pure toggles, represented as a line containing nothing but hyphens (at least three of them), that allow the programmer to signify that they are entering or exiting a piece of code where they want the absence of side-effects to be strictly enforced. Rabbit also provides a function, pure, that will execute the code it is given inside of a pure block. These utilities allow programmers to take full advantage of the benefits of referential transparency without having to sacrifice ease of input/output.

Instead of providing a full specification of each of Rabbit's data types and operators—which can be found in the full Technical Whitepaper—this paper will briefly touch on the purpose and syntax of each data type before going into examples. Rabbit shares many data types with Python, but also uses many of its own, and uses different syntax for all of them—although an attempt was made to make the basic syntactical constructs as similar as possible to existing Python ones so as to ease the process of learning the language. Because Rabbit is a functional programming language, all of its data types are immutable. Integers, numbers, Booleans, and strings are much the same as in Python, as well as the operations on them. Conditionals are represented using Rabbit's at-symbol semicolon syntax, where the conditionals—represented as the intended result if the condition is true, then an at-symbol, then the condition—are separated by semicolons and checked in order. Rabbit mainly uses two containers, the row and the list, with the row created much like Python's list and the list created much like Python's tuple. Functions are created in the normal mathematical way—using parentheses and commas—and can be called that way or by following the function with colon and separating the arguments by colons. Functions can also be created in-line using lambda syntax, where the function variables are set off by backslashes and the function body follows the closing backslash. A special operator, the tilde, functions like an enhanced version of the map construct, applying a function, the rightmost operand, to every item in a list. Dictionaries in Rabbit function similarly to in Python, but use an arrow instead of a colon to separate keys and values. While Rabbit is mostly a functional programming language, it also supports object-oriented features, and has classes and instances. Classes make possible the Rabbit where clause, which uses the dollar-sign operator to allow variable assignments in the right operands which should apply only to the evaluation of the leftmost operand. Additionally, Rabbit has the ability to use its wrap object to allow it to interface with any arbitrary Python object, a tool used by the Rabbit interpreter to allow it to make direct use of Python's import statement, granting Rabbit direct access to the entire Python standard library. Finally, Rabbit also has its own built-in standard library, which provides extensive utilities for higher-order functions, meta-programming, and high-level mathematics.

Rabbit's syntax allows for powerful operations to be performed in minimal lines of code. The classic example given to prove the power of functional programming languages is the programming of the quick sort function, a complicated sorting algorithm that combines the first element of a list, the sorted version of all the elements in the list less than or equal to that element, that element, and the sorted version of all the elements in the list greater than that element. In an imperative language, even one with clean, simple syntax like Python, the preferred algorithm is long and hard to easily understand:[17]

```
def quickSort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

Haskell, by providing functional programming features, significantly simplifies the code. But by tacking on type information and using multiple function definitions, the preferred Haskell code is still longer and more cumbersome than would be desired:[18]

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Rabbit, on the other hand, manages to achieve the shortest, simplest, most concise function definition:

[17] The source for the Python quick sort implementation is: http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Python
[18] The source for the Haskell quick sort implementation is: http://learnyouahaskell.com/recursion#quick-sort

```
qsort(l) = (
    qsort: (as ~ \x\(x @ x<=a)) ++ a ++ qsort: (as ~ \x\(x @ x>a))
    $ a,as = l
    ) @ len:l
```

A commented, more spread out version of that code explains what is happening:

```
# The quick sort function:
qsort(l) = ( # qsort is defined as a function of one argument, l
        qsort: # The qsort function is called recursively on all the
               #  elements in the list less than or equal to the pivot.
            (as ~ # The tilde operator loops over the list with a function
                 \x\( # An in-line function definition using backslashes
                     x @ x<=a # The main body of the function
                     ))
        ++ a ++ # That sort is joined with the pivot, which is then joined
                #  with the next sort.
        qsort: # The qsort function is called recursively on all the
               #  elements in the list greater than the pivot.
            (as ~ # The tilde operator loops over the list with a function
                 \x\( # An in-line function definition using backslashes
                     x @ x>a # The main body of the function
                     ))
        $ a,as = l # The original argument, l, is split up into two parts,
                   #  its head, a, which will be used as the pivot, and its
                   #  tail, as, which will be the list that is split up
                   #  into two parts and each part sorted.
        ) @ len:l # The whole body of the function is only performed if l
                  #  is not empty, otherwise null, the empty list, is
                  #  returned.
```

When expanded as such, the Rabbit algorithm looks like it has become as long as the Python one. But the Python algorithm documented as extensively as this piece of Rabbit code would be much longer. Thus, Rabbit shows its use of functional programming in its ability to allow for much shorter, much more concise code.

III. Discussion

Rabbit isn't finished. The writing of this paper marks the release of Rabbit 1.0.0,[19] but despite finally reaching its first production release, greater than 40 GitHub issues still remain as

[19] That release can be accessed at: https://github.com/evhub/rabbit/releases/tag/v1.0.0

of this writing, mostly features or enhancements that the author intends to add into the language soon.[20] Many of these will help make Rabbit faster, more modern, and more extensible. Some large outstanding missing features include the ability to pass keyword arguments, a data type for mathematical sets, and the ability to redefine and add new operators. But the three most important are lazy lists, smarter atoms, and support for parallel computing. Each one of these will take a great deal of work, but specifications have already been drawn up to implement each of them. Lazy lists will mean the close integration of Rabbit matrices with Python generators to prevent list values from being calculated until they're needed, allowing for infinite lists, and making finite lists much faster. Smarter atoms will mean atoms that remember the operations that were done to them, and only compare true to items that could have had those same operations done, greatly increasing Rabbit's pattern-matching utility. Finally, support for parallel computing will mean the addition of features—such as futures and promises—that allow for easy integration with multiprocessing and multithreading features, as well as, because Rabbit's functional nature allows for it, parallelization of much of the interpreter, including when lazily assigned variables are evaluated. This will greatly increase the speed of the language.

The Rabbit language, even in its current state as it is presented in this paper, has already started to show its practical utility. Because of the ease of extending the language in Python, the author wrote a variety of helper programs that make use of Rabbit. A program to graph arbitrary Rabbit functions was written, and has been used by the author for a variety of demonstrations on scientific or mathematical topics.[21] A basic Integrated Development Environment with support

[20] The full list of issues can be found at: https://github.com/evhub/rabbit/issues
[21] The code of this program can be seen in rabbit/graph.py and the program accessed through rungraph.py in the Rabbit repository.

for syntax highlighting and running code was written to help Rabbit developers.[22] To demonstrate the ease of writing Rabbit and hooking into Rabbit, a repository containing an example workspace was created.[23] And to show how Rabbit can help children learn math, a program was written that challenges the user with math problems and provides a Rabbit interpreter to solve them with.[24]

Finally, this paper envisions even more possible uses for Rabbit in the future. Rabbit's powerful meta-programming features and easy extensibility with Python make it perfect for creating DSLs, or Domain-Specific Languages. Domain-Specific Languages are specialized languages made for very specific purposes. These include, but are not limited to, controlling robots, engineering tools, and scientific instruments. An example domain-specific implementation of Rabbit was created by the author for the much more casual purpose of controlling characters in a Dungeons and Dragons strategy game, but the program is simply a proof-of-concept for the use of Rabbit to build a Domain-Specific Language for a specific purpose.[25] The creation of new Domain-Specific Languages within Rabbit represents another possible avenue for future work on the language as well as highly tangible, useful application of the language. It is hoped that Rabbit will prove an effective tool for integrating the benefits of functional programming into Domain-Specific Languages as well as bringing the benefits of functional programming to the average, everyday programmer.

[22] The code of this program can be seen in rabbit/ride.py and the program accessed through runride.py in the Rabbit repository.
[23] The repository for this program can be found at: https://github.com/evhub/rabbit-environment
[24] The repository for this program can be found at: https://github.com/evhub/mathtest
[25] The repository for this program can be found at: https://github.com/evhub/rpgengine

References

"A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering," Hans Fangohr, Computational Science - ICCS 2004, Lecture Notes in Computer Science Volume 3039, 2004, pp. 1210-1217, link.springer.com/chapter/10.1007/978-3-540-25944-2_157.

"A Gentle Introduction to Haskell," Paul Hudak, John Peterson, Joseph Fasel, 2000, haskell.org/tutorial/.

"Computer Programming for Everybody," Guido van Rossum, July 1999, funding proposal submitted to DARPA, python.org/doc/essays/cp4e/.

Foreword written by Guido van Rossum for "Programming Python," Mark Lutz (1st ed.), 1996, python.org/doc/essays/foreword/.

Journal of Functional Programming, Matthias Felleisen, Jeremy Gibbons, Editors, journals.cambridge.org/action/displayJournal?jid=JFP.

"Learn You a Haskell for Great Good," Miran Lipovača, 2011, learnyouahaskell.com/.

Programming Language Popularity, October 25, 2013, http://langpop.com/.

"PYPL PopularitY of Programming Language Index," PyDatalog, sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language.

"Python as a Programming Language for the Introductory Programming Courses," Jussi Pekka Kasurinen, Thesis for the Degree of Bachelor of Science in Technology, Lappeenranta University of Technology, 2006, pdf.aminer.org/000/278/636/a_comparison_of_c_matlab_and_python_as_teaching_languages.pdf.

"Real World Haskell" Bryan O'Sullivan, Don Stewart, and John Goerzen, 2008, book.realworldhaskell.org/.

Sorting Algorithms/Quicksort, wiki page, rosettacode.org/wiki/Sorting_algorithms/Quicksort#Python.

Stack Overflow, "Are there any decent scripting languages that use functional programming," stackoverflow.com/questions/733222/are-there-any-decent-scripting-languages-that-use-functional-programming.

"Talking About Language," Janet Swift , August 27, 2014, i-programmer.info/news/98-languages/7684-haskell-most-talked-about-language.html.

Teaching an Introductory Computer Science Sequence with Python," Bradley N. Miller, David L. Ranum, micsymposium.org/mics_2005/papers/paper20.pdf.

"Teaching Programming with Python and PyGame," Vern Ceder, Nathan Yergler, tech.canterburyschool.org/pycon/teaching_pygame.pdf.

TIOBE Index for September 2014, tiobe.com/index.php/content/paperinfo/tpci/index.html.

"Top 10 Programming Languages," July 19, 2014, IEEE Spectrum, spectrum.ieee.org/computing/software/top-10-programming-languages.

"Why Complicate Things? Introducing Programming in High School Using Python," Proceedings of the 8th Australasian Conference on Computing Education, Volume 52, pp. 71-80, Australian Computer Society, Inc., 2006, dl.acm.org/citation.cfm?id=1151880.

"Why Functional Programming Matters," John Hughes, from "Research Topics in Functional Programming" ed. D. Turner, Addison-Wesley, 1990, pp. 17–42, cs.utexas.edu/~shmat/courses/cs345/whyfp.pdf.

"Why New Programming Languages Succeed or Fail," InfoWorld, Neil McAllister, March 15, 2012, infoworld.com/article/2610284/application-development/why-new-programming-languages-succeed----or-fail.html.

"Yet Another Haskell Tutorial," Hal Daume III, 2002-2006, umiacs.umd.edu/~hal/docs/daume02yaht.pdf.