YaleNUSCollege

Lisp in the 21st Century: A Study of Paul Graham's Bel Programming Language

Ryan Foo Mao Yao

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Professor Olivier Danvy

AY 2020/2021

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

   [✓ o Unrestricted access]
   Make the Thesis immediately available for worldwide access.

   o Access restricted to Yale-NUS College for a limited period
   Make the Thesis immediately available for Yale-NUS College access only from _____________ (mm/yyyy) to _______________ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
   ____.

   After this period, the Thesis will be made available for worldwide access.

   o Other restrictions: (please specify if any part of your thesis should be restricted)
   ____
   ____

Ryan Foo Mao Yao, Cendana
Name & Residential College of Student

[illegible]
Signature of Student

April 5th, 2021
Date

Professor Olivier Danvy O. Danvy
Name & Signature of Supervisor

April 5th, 2021
Date

Acknowledgements

I would like to thank the following people.

Professor Olivier Danvy, my capstone advisor, for his interest in my Computer Science education. His patient efforts helped shape my understanding and knowledge of the mathematical and computational sciences, and inspired me to learn more about computing (and Lisp).

My friends, both in and out of 12-505, for the great moments we shared in this last year, and specifically Alaukik and Leyli for helping proof-read this report.

My loved ones, for supporting me through this journey.

Thank you.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Lisp in the 21st Century:

A Study of Paul Graham's Bel Programming Language

by Ryan FOO

Lisp is a programming language that ignited a revolution from numerical computing to symbolic computing. We are interested in studying its impact on programming languages today. We present a brief history of Lisp and explain the concepts of an axiomatic phase and an implementation phase in relation to programming language design. We apply these concepts to understand Paul Graham's new dialect of Lisp, Bel. Using Python, we implement a properly tail-recursive interpreter for a subset of Bel, which we title Babybel. We provide a REPL and a test suite for Babybel, as well as an analysis of sample programs.

Contents

Chapter 1

Introduction and Motivation

"We first describe a formalism
for defining functions recursively."

John McCarthy

1.1 Lisp and The Programming Language Landscape

1.1.1 What is Lisp?

LISP is one of the oldest programming languages still in use today, next to FORTRAN. Lisp was formulated by John McCarthy at MIT and is documented in the paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine" [15]. McCarthy did something remarkable for computing: they showed that one could build a Turing-complete language that operated on symbols using a few operators and a notation for denoting functions inspired by Alonzo Church's lambda calculus [4].

Lisp took a radical approach to computing, and is characterised by the following ideas:

- "computing with symbolic expressions rather than numbers."
- “representation of symbolic expressions and other information by list structure in the memory of a computer.”
- “a small set of selector and constructor operations expressed as functions.” [14]

Most of these ideas were novel at the time, compared to other programming languages such as FORTRAN. Lisp introduced concepts such as conditionals, functions as first-class objects, recursion and garbage collection into programming [12].

1.1.2 The Lisp Revolution

Lisp symbolised a revolution from numerical to symbolic computing: rather than modifying values in containers, we were evaluating symbolic expressions. This allowed Lisp programmers to stop struggling with encodings and representations (the "how") and instead compute on what was encoded or represented (the "what").

The most telling example is memory management: in FORTRAN, one has to allocate and deallocate memory (malloc) when writing their program. However, Lisp internalized the task of memory management with a garbage collector, freeing up programmers to reason about their programs.

1.1.3 Why Lisp?

Lisp’s main appeal to programmers is its extreme flexibility: Joel Moses called it a “ball of mud… add more to Lisp, and it’s still a ball of mud — it looks like Lisp.” The reason it looks like Lisp ("Lots of Silly Parentheses") is because of all its parentheses, which give a uniform syntax, and allow programmers to easily add new language features [3]. Macros in Lisp allow one to have control over the language syntax by manipulating the abstract syntax tree [20].

That flexibility and malleability allows Lisp programmers to write programs and features extremely fast — Paul Graham suggested that Lisp was a “secret weapon”, and that Lisp tremendously sped up their development cycle, such that they could easily clone competitors’ features [9].

1.1.4 The Evolution of Lisp

According to Steele and Gabriel, Lisp made it easy to “extend the language or even to implement entirely new dialects from scratch.” [20] This flexibility is enabled by macros and Lisp’s structure: because both code and data are represented using lists, the language implementation can be overwritten during run time.

To that end, many Lisp dialects have been implemented, to various degrees of success. Scheme, for example, was implemented by Steele and Sussman, as a dialect with lexical scope and proper tail recursion [21]. Scheme was used for introductory computer science education at more than 20 schools through the 1980s and 1990s, including MIT [22].

In the 1980s, there was an effort to standardize Lisp for common use – leading to the development of Common Lisp in 1984. American National Standards Institute (ANSI) Common Lisp was the result of a decade long effort to standardise the language [2]. However, this did not lead to widespread adoption of Common Lisp today [19].

1.1.5 Motivations

I was personally interested in developing a finer understanding of the power of Lisp from Professor Olivier Danvy and Paul Graham. I desired to study Bel in depth precisely because I wish to understand the expressive capability of Lisp programming languages and have a "profound enlightenment experience... (that) will make (me) a better programmer for the rest of (my) days" [17].

The first step for me was beginning to study this new dialect of Lisp, named Bel, which was designed by Paul Graham.

"Lisp is winning again."

Opening banner in the REPL of classical Lisp implementations

Chapter 2

Paul Graham's Take on Lisp

2.1 Who is Paul Graham?

Paul Graham is a wise computer scientist who built his first startup, Viaweb, using Lisp. Viaweb later was sold to Yahoo Stores for about 49 million USD. Paul Graham attributes much of Viaweb's success to their secret weapon: writing in Lisp [9]. They have written two books on Lisp: On Lisp (1993) and ANSI Common Lisp (1995). They are more commonly known as one of the co-founders of Y Combinator, a highly influential start-up accelerator which invested into Stripe, Airbnb, Coinbase, Twitch, Reddit and Volley, amongst other investments [25].

In October 2019, Paul Graham came up with a new specification for a dialect of Lisp, named Bel [10].

2.1.1 Why is Paul Graham's implementation interesting?

Paul Graham claims that what made Lisp interesting was the fact that its development happened in two parts: a formal phase, represented by the 1960 paper by McCarthy, and the implementation phase, where the "language was adapted and extended to run on computers. [11]"

Their hypothesis was that it's probably a good thing that we defined the language by writing an interpreter for the language in terms of itself, and that fact is "responsible for a lot of Lisp's best qualities [11]."

2.1.2 The Axiomatic Approach to Lisp

According to Paul Graham, the axiomatic approach to Lisp is the approach towards Lisp defined in the seminal 1960 paper by McCarthy. The paper describes a notation, or a "formalism for defining functions recursively", which "has advantages both as a programming language and a vehicle for developing a theory of computation" [15].

This notation was based on Alonzo Church's lambda calculus, which was proven to be a Turing-complete universal model of computation [23]. The formal phase produced a specification for the programming language: what functions it must implement, what syntax it uses, and what are the different data types that exist in the language. This is the "what" of the language.

This was separate from the implementation phase, where enterprising computer scientists went on to, given this specification, implement a fully functional programming language that can be run on computers, or what we will call an operationalised version of a programming language. In this phase, computer scientists select a meta-language in which to implement the specification. In our case, we have chosen Python to be our implementation language – and this is the "how" of the language.

This notion of the axiomatic approach and the operationalised version of Lisp, or alternately, the formal phase and the implementation phase, will feature heavily in our discussion. So if Paul Graham has provided the "What" for Bel, then we are taking steps about the "How".

The Bel that Paul Graham specifies is purely a product of the formal phase, and is a thought experiment where Paul Graham, an expert Lisp programmer with more than 25 years of experience, aims to "delay (the) switch" from the formal to the implementation phase "for as long as possible [11]."

We were interested in investigating this question, and implementing an interpreter for a limited version of Bel, which we name Babybel, after the French cheese (Figure 2.1). Allow us to distinguish between Paul Graham's specification (Bel) and the interpreter we have implemented (Babybel), and consider Babybel our contribution to the Tower of Babel of programming languages.

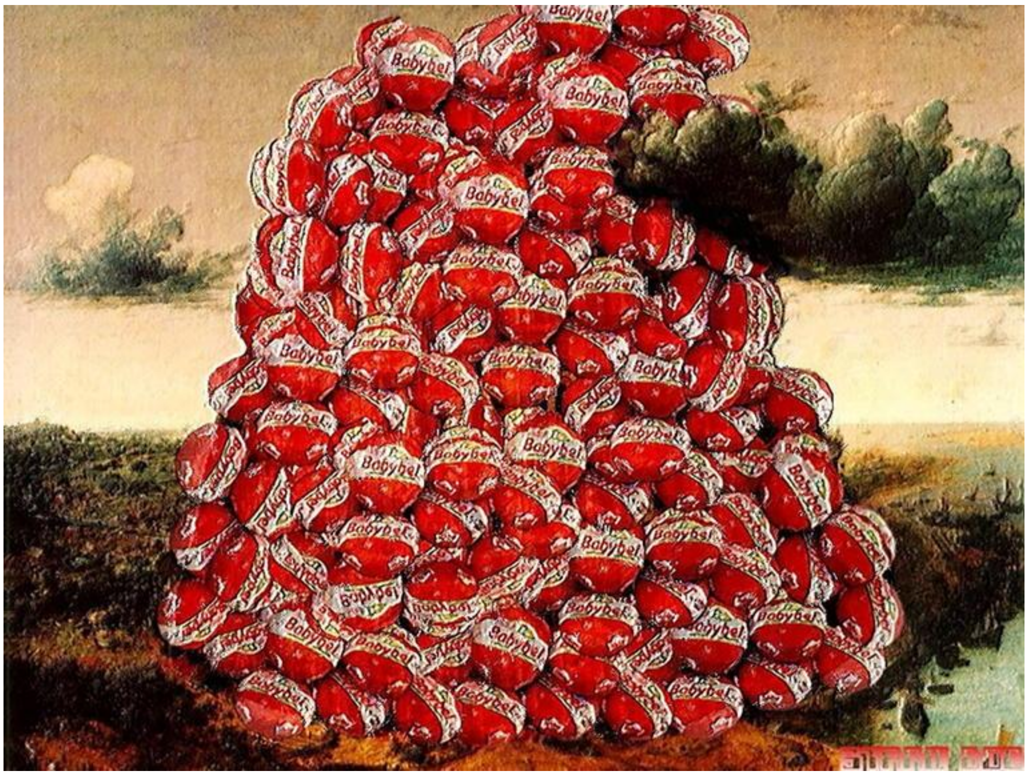


FIGURE 2.1: A tower of Babybel, after Bruegel. (Twitter)

Chapter 3

The Implementation Phase of Bel

3.1 Operationalizing Bel as Babybel

While Paul Graham's lyricism about Bel is beautiful, to our knowledge, Paul Graham is yet to publicly publish an interpreter for Bel. If indeed, they wanted to write their own ultimate version of Lisp, it may be interesting then to take up that challenge and implement Bel, as well as write and explain some programs with Bel. To that end, we wanted to operationalize Bel.

In this chapter, we detail our implementation of a subset of Bel (without macros and streams), titled Babybel. We shall demonstrate how we have transformed Paul Graham's axiomatic description of Bel into an operational version of Bel.

3.1.1 Literate Programming

We attempt to write this chapter as a literate programming document –

We understand literate programming as divided in two processes: weaving and tangling.

Weaving is the generation of this capstone document, and particularly this chapter, which details the implementation of the interpreter.

Tangling is the execution of the development of this interpreter for Bel, or Babybel, which has already occurred by the time of this writing. This tangle can be found as a repository on GitHub that one can clone to one's computer and a functioning REPL that can be run on one's command line, with a set of test programs included. (Appendix A)

We will consider our implementation successful if we can run a subset of Bel programs using our read-eval-print-loop (REPL) and attain correct Bel results. This involves the understanding and development of readers, interpreters, primitives, axioms, environments, and functions to navigate those. It also involves scoping down the goals of our implementation, and of course, parsing the Bel language specification by Paul Graham.

At risk of appearing stylish, inline Python code snippets will be colored `olive`, whilst inline Bel snippets will be colored `blue`. This will help us remain clear-minded about which functions exist at which layers of implementation.

3.2 On Readers and Interpreters

A reader and interpreter are responsible for helping a user go from Babybel program to the output on their console.

We implement Babybel expressions as Python objects, and inherit and modify the reader for reading Lisp expressions from William Annis [1].

3.2.1 Dear Reader...

We modify the reader from William Annis's PyLisp, in the process adapting it to Python 3.9 language syntax [1]. We also modify it to add our Number type as another axiomatic expression.

Typically, a parser is composed of a lexer and the proper parser. However, our reader is a scannerless parser, which means the two steps are inter-weaved. Our reader begins building up the abstract syntax tree by moving sequentially through the provided Python string.

Our reader works as follows: given a Python string, which represents a Bel expression, the reader transforms this Python string into a Bel expression, which can then be evaluated. Along the way, it decides based on a set of rules whether the token is a Bel Symbol, a Bel Number, parentheses or a quote, and constructs a list that contains these elements accordingly. The constructed Bel expression, or an Abstract Syntax Tree (AST), can then be passed to `eval`.

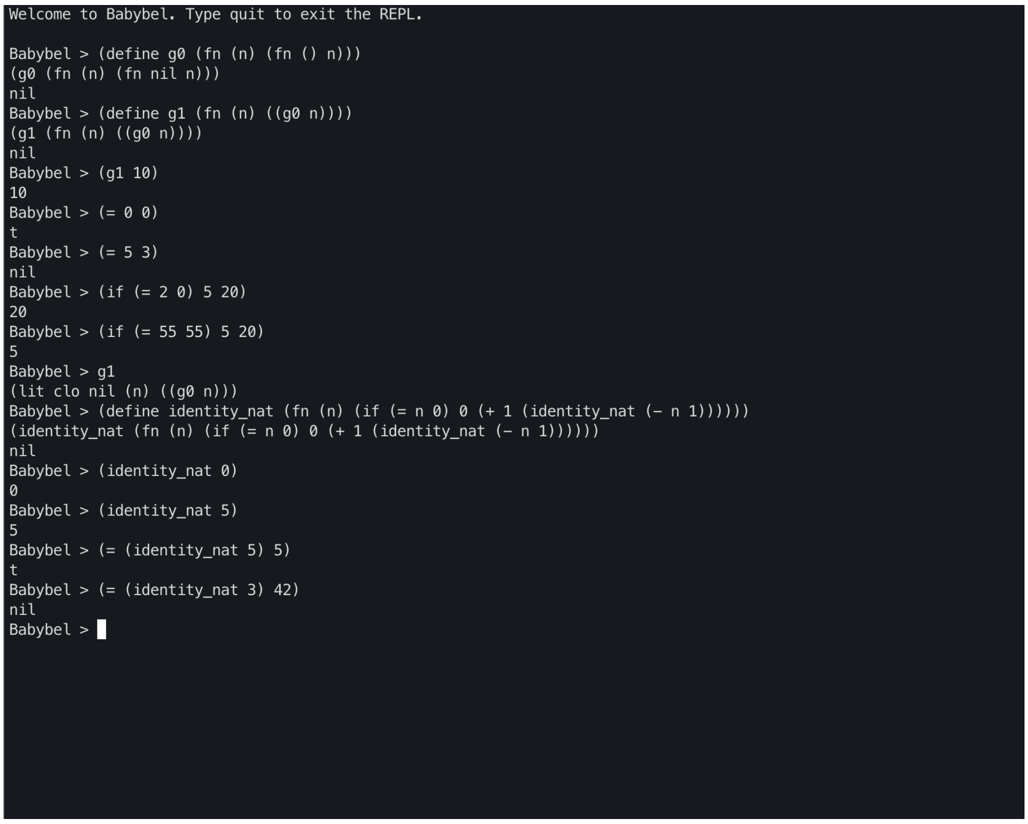
3.2.2 Read-Eval-Print-Loops (REPL)

A read-eval-print-loop reads a user's input and parses it into an AST, evaluates that AST and returns a new AST, prints that AST to the console, and repeats the process, asking them for input again.

In our specific case, it reads a Python string using our reader, evaluates it using our interpreter (`eval`), and then it prints the output of the evaluation to the console.

The user can also define variables, which will be added to the environments for the duration of the REPL session. Future work may look at implementing reading programs from files with .bbel extensions, in order to implement persistence.

We include some non-standard additions: typing `quit` will exit the REPL session. Additionally, typing `(g_env)` will return the user the current global environment as a Python dictionary. These additions assisted a great deal in debugging the REPL.



```
Welcome to Babybel. Type quit to exit the REPL.

Babybel > (define g0 (fn (n) (fn () n)))
(g0 (fn (n) (fn nil n)))
nil
Babybel > (define g1 (fn (n) ((g0 n))))
(g1 (fn (n) ((g0 n))))
nil
Babybel > (g1 10)
10
Babybel > (= 0 0)
t
Babybel > (= 5 3)
nil
Babybel > (if (= 2 0) 5 20)
20
Babybel > (if (= 55 55) 5 20)
5
Babybel > g1
(lit clo nil (n) ((g0 n)))
Babybel > (define identity_nat (fn (n) (if (= n 0) 0 (+ 1 (identity_nat (- n 1))))))
(identity_nat (fn (n) (if (= n 0) 0 (+ 1 (identity_nat (- n 1))))))
nil
Babybel > (identity_nat 0)
0
Babybel > (identity_nat 5)
5
Babybel > (= (identity_nat 5) 5)
t
Babybel > (= (identity_nat 3) 42)
nil
Babybel >
```

FIGURE 3.1: This is a screenshot of a REPL session with the repl.py program, which uses the reader and evaluator. Witness demonstration of the identity function, defined as a Bel function.

3.3 Implementing the Bel Language

We start by defining Bel expressions as Python objects. This allows for inheritance of methods and attributes. Since we are more concerned with rapid development speed and conceptual understanding, concerns of memory and speed here are secondary to conceptual clarity. (Worse-Is-Better in action [7].)

3.3.1 Types in Bel

Every type in Bel is implemented in Python as a `BelType` object.

Bel types inherit from `BelType`, and are either a `Char`, `Pair`, `Symbol`, `Stream` or `Number`.

Numbers are non-standard but implementing them makes our job easier – the ability to reuse Python's number system of `ints` and `floats` allows us to abstract a lot of the complexity of implementing fractions, to focus on more illuminating topics.

3.3.2 Atoms

Atoms are a foundational type in Bel. All Bel types except Pairs are atoms.

Symbol

`Symbol`, or symbols, are words. Furthermore, one can instantiate them and look them up on the symbol table, which provides O(1) lookup time for bound symbols.

We implement symbols as Python objects with names, and implement the symbol table as a Python dictionary.

Char

`Char`, or characters, are implemented as Python objects with the attribute of name. We omit the backslash character from Babybel due to the difficulty of contending with Python's use of \as an escape character.

Streams

Babybel does not implement Bel's specified streams, as we chose to focus and build up the tools for functional programming within Babybel. However, we acknowledge that streams are used for input and output in the Bel language.

3.3.3 Pairs

Pairs are the first compound type in Bel, and serve as the building block for more complex data structures. They are made of two other `BelTypes`.

In Bel, we represent them as such: `(foo . bar)`. This is a pair of the symbols, `foo` and `bar`. On the Python layer, we represent the same pair as `Pair(Symbol(foo), Symbol(bar))`.

3.3.4 Lists

We implement lists using pairs. In the business, we call them association lists, because one associates a variable (the head) with a value (the tail). In the case of Bel, we represent nodes using Bel pairs, where the `car`, or the head of the list, is the variable, and the `cdr`, or the tail of the list, is the rest of the list.

The symbol `nil` represents the empty list, and if `y` is a list, then the pair `(x . y)` is a list of `x` followed by the elements of `y`. [10]

We follow an algorithm for getting lists to represent themselves in abbreviated list notation (1 2 3) rather than in dot notation. The algorithm is as follows: [10]

1. The symbol `nil` can also be represented as `()`.

2. When the second half of a pair is a list, one can omit the dot before it and the parentheses around it. So `(a . (b ...))` can be written as `(a b ...)`.

By applying these rules, one can transform `(a . (b . (c . nil)))` into `(a b c)`, which is how we represent lists in practice.

Strings

Strings are lists where all the args are characters.

3.3.5 Helpful Predicate Functions

There exists a continuous discourse between the implementation layer (Python) and the language layer (Babybel). We implement predicates in Python that help us to debug the interpreter in the language layer, such as `atomp`, which given a Bel object, checks if its an atom.

These predicates prove helpful as they provided a base for implementation of some of the more complex functions.

Examples of such predicates were `symbolp`, which tells us if a Bel expression is a symbol, `proper_listp`, which tells us if a Bel expression is a proper list. These are Python programs that can be used in implementing our larger programs such as `eval` and `function_apply`.

It helped us to have various methods of transforming data from Python types into Bel types, defined both as object initialization methods and as stand-alone functions.

Consider the `make_list` function, which constructs a Bel list out of a list of arguments. This allowed us quick construction of proper lists, simplifying the bootstrapping process.

```
def make_list(*args):
    if (len(args) <= 0):
        return bel_nil

    base_pair = Pair(args[-1], bel_nil)
    result_list = base_pair

    for i in range(len(args)-2,-1,-1):
        result_list = Pair(args[i], result_list)

    return result_list
```

3.3.6 Environments

Bel specifies the existence of a lexical, dynamic and global environment. The goal of an environment is to "represent names and their denotation [6]."

We refer to the name, as the variable, and the value as the denotation. An environment is implemented as a Bel list of Bel pairs, and each pair of `(variable . value)` is the binding of a symbol `var` to its value `val`.

One can look up a symbol `var` in the environment and retrieve the value `val`. One can add new pairs to the environment at any time, with either push or bind. `push` uses only one pair whilst `bind` can handle multiple var-val pairs.

We can look up environments given a symbol, and this is done by traversing the dynamic environment, then the lexical environment, then the global environment. This corresponds to the Babybel function `lookup`.

We instantiate the global environment as a hash table, which allows us to look up in O(1) time. The dynamic and lexical environments, in comparison, are instantiated on run time as association lists (because we extend them within Babybel).

In Babybel, we implement the global environment as a Python dictionary, `symbol_table`, and the dynamic and lexical environments are implemented as instances of a Python object `Env`, which represents a Bel list of pairs.

3.3.7 Bel Axioms

When it comes to axioms in Bel, we think it best to quote Paul Graham:

> "Like McCarthy's Lisp, Bel is defined starting with a set of operators that we have to assume already exist. Then more are defined in terms of these, till finally we can define a function that is a Bel interpreter, meaning a function that takes a Bel expression as an argument and evaluates it.

> There are two main types of axioms: primitives and special forms. There are also a few variables that come predefined."

It's fair to say then that these axioms are the bedrock of the Babybel language, and that we are responsible for using our meta-language to implement them.

Predefined Variables

These are the predefined variables in Bel.

`nil`: the empty list, or falsity.

`t`: The default symbol for truth.

`lit`: The symbol that signifies that an object evaluates to itself. (Applying `eval` to the Bel object will return one that self-same object.)

In addition, we will associate a set of variables with values that indicate they are primitives. For example, we associate the symbol `+` with `(lit clo +)`, and register it on the global environment.

3.3.8 Literals and Primitives

Literals are Bel objects that evaluate to themselves.

They are seen in the form `(lit . rest)`, where lit is a symbol, and rest is a proper list of Bel objects.

Primitives are functions that are assumed to exist for a self-interpreter. Therefore, these are the functions that we are expected to implement using our meta-language.

Primitives and functions are literals.

We develop a function to generate them: `make_literal` creates a pair where the `car` is the symbol `lit`, and the `cdr` is anything that should be treated as a literal.

We also develop means to make primitives and register them in the environment. We initialize the environment with said primitives, so that looking up the symbol `("+")`, for example, will return the list `(lit prim +)`, where all of the elements are symbols. This will be handled accordingly by `eval`.

We chose to implement a subset of those primitives for Babybel. This subset includes `id, join, car, cdr, type, xar, xdr, sym, coin, +, -, *, /, <, <=, >, >=, =, set, define`. We include also the primitive `g_env`, which is non-standard but allows us access to the symbol table while programming in Bel. We chose the primitives that are directly relevant to functional programming, and have left out those related to programming with streams and bits, as we are focused on alternate streams of knowledge.

Consider `make_primitive` and `register_primitive_in_env`. These functions do as their names suggest. The latter registers a primitive on the global environment.

```
def make_primitive(symbol):
    return make_literal(Pair(bel_prim,
                             Pair(symbol, bel_nil)))


def register_primitive_in_env(x):
    symbol = Symbol(x)
    symbol_table[x] = make_primitive(symbol)
```

When one looks up these symbols, for example, `+`, one is returned with `(lit prim +)` in the symbol table. The environment has been updated by `generate_primitives`, which registers these primitives. Upon evaluation, we are brought to the primitive branch in `eval`, which then tells Python to call the implemented primitive functions.

3.4 Functions

Functions can take a variadic number of arguments, do something with those arguments (or nothing), and then return one expression.

They are represented using lists. This is a function in Babybel.

```
(lit clo nil (x) (+ x 1))
```

The keyword `lit` tells the interpreter that the function is not to be evaluated. The keyword `clo` tells you that it is a closure.

The third argument, `nil`, is the local environment, (or lexical environment), a list of variables that have values from being parameters of previous functions. In the example above, the environment is the empty environment, which means there are no bindings.

The fourth argument, `(x)`, is the function parameters, or the formal parameters. They will be replaced by whatever one calls the function with.

The fifth argument, `(+ x 1)`, is the body of the function, and it defines what value the function will return upon application.

In practice, we would express a function using `(fn (x) (+ x 1)`. Evaluation of that expression would yield our function `(lit clo nil (x) (+ x 1))`, which can then be applied.

3.4.1 Closures

A function with a lexical environment is called a closure. If a closure includes an environment with a bound value for `x`, as in the example above, then `x` will have a value within the closure, or the function with that lexical environment.

To make a closure, we take an environment and a list. The list has two elements: the formal parameters, and the body of the function.

```
def make_closure(l_env, rest):
    return make_literal(Pair(bel_clo, Pair(l_env, rest)))
```

3.5 Evaluation

Evaluation represents the capstone of a Lisp interpreter, and it promises to provide us a base to run most arbitrary programs in that language.

3.5.1 On Evaluators

We will study two functions: `eval`, and `function_apply` (function application). We pass Bel expressions, with a lexical environment, to eval, which will call themselves until evaluation is complete, and have special forms (special cases) that will implement our desired interpreter.

We implement `eval` and `function_apply` in tail recursive style, which has the advantage of generating a minimal call stack

Eval, or the evaluation function, takes a Bel expression and evaluates it accordingly.

The beauty of the evaluation function in Bel is that the fundamental data types are well defined. Therefore, the language remains lean, while maintaining maximum expressivity. This aligns with the Lisp philosophy of giving additional expressive power to the programmer, abstracting away the "How". Every program that a programmer writes will either operate on atoms, pairs, or conforms to a special form.

The power of Lisp is that because the evaluator is so tightly bounded and axiomatically expressed, one can morph Lisp to have any language features they desire by extending the language to fit their own needs, such that they can then export that language and run it on any computer.

This is the same phenomenon that causes some programmers to become so attached to their emacs bindings, and programs that they've written to manipulate symbols in a text-editor. They have extended their environment WITHIN their tools, which is a tremendous power to impart upon a budding programmer.

For Lisp, and Babybel, the evaluator is where all the magic happens.

3.5.2 Eval

The version of `eval` we implement is as follows:

```
def eval(exp, l_env):
    if numberp(exp):
        return exp
    elif symbolp(exp):
        if (idp(exp, bel_nil) or idp(exp, bel_t) \
        or idp(exp, bel_o) or idp(exp, bel_apply)):
            return exp
        else:
            return lookup(l_env, exp)
    elif quotep(exp):
        return special_quote(exp, l_env)
    elif literalp(exp):
        return exp
    elif stringp(exp):
        return exp
    elif pairp(exp):
        if idp(exp.a, Symbol("fn")):
            return make_closure(l_env, exp.d)
        elif idp(exp.a, Symbol("if")):
            return eval_tp(exp, l_env)
        elif idp(exp.a, Symbol("set")):
            return special_set(exp.d, l_env)
        elif idp(exp.a, Symbol("define")):
            return special_set(exp.d, l_env)
        else:
            return function_apply(eval(exp.a, l_env),
                         eval_list(exp.d, l_env))
    raise Error("%d is not a proper list,
    you cannot apply a function." % exp)
```

Let us take a first parse of this program, before going into a deeper analysis.

Every expression is either a pair, or an atom.

If it is an atom, check if it's a number.

Else, if it's a symbol, then look that symbol up in the symbol table. If they are an axiom symbol, then they evaluate to themselves. (the `nil` symbol, or the `t` symbol, for example.) Otherwise, look them up in the environments, starting from the dynamic environment, moving to the lexical environment and ending in the global environment.

If it is not an atom, then it is a pair. From there, we can use our other predicates to determine if it's a string, or a literal, or a quote, among other special forms and meta-level functions we choose to define.

This is the expressive power of Lisp beginning to manifest.

If the expression is a special form, a proper implementation of the `eval` program will have a branch that handles a different way to handle the expression's evaluation.

3.5.3 Basic Cases

Evaluating Numbers and Symbols

If it is a singleton, we use the predicates `numberp` and `symbolp` to determine the type of the expression. Numbers evaluate to themselves. If the singleton is a symbol, then look that symbol up in the symbol table. If they are an axiom symbol, then they evaluate to themselves. (the `nil` symbol, or the t symbol, for example.) Otherwise, look them up in the environment, starting from the dynamic environment, moving to the lexical environment and ending in the global environment. If they aren't there, return `nil`.

Evaluation of Primitives

Evaluating a primitive is simple. We extract the symbol, for example, `+` from `(lit prim +)`, and apply it to the `args`. Recall that `args` is a Bel list. We perform an arity check, fetch the evaluated arguments to the function from the args list, and apply the necessary depending on what the function specifies.

This behaviour is implemented in the various functions, which, depending on the primitive, are called during evaluation by apply_primitive.

Witness a selection from apply_primitive:

```
def apply_primitive(sym, args):
    if lookup_prim(sym, "id"):
        return prim_id(args)
    elif lookup_prim(sym, "join"):
        return prim_join(args)
    # ...
    elif lookup_prim(sym, "type"):
        return prim_type(args)
    # ...
    elif lookup_prim(sym, "+"):
        return prim_add(args)
    elif lookup_prim(sym, "-"):
        return prim_sub(args)
    elif lookup_prim(sym, "*"):
        return prim_mul(args)
    else:
        raise ValueError("Unknown Bel Primitive.")
```

We deviate from Paul Graham's Bel specification as we include primitives for operations such as addition, comparison ($\geq$, $\leq$) and other operations on numbers, which are non-standard for Bel.

We also leave a few of the programs unimplemented, such as `dyn, where, after`, because we scoped the language down to a smaller version of Bel (Babybel). This is clear if one is to take a look at Appendix A, which contains the full Babybel source code.

Example: Identity Primitive

Consider this program, `(id 5 5)`. The first element of this list, `id` is a primitive, which takes two arguments and checks whether both arguments are identical (this is stricter than equality).

Our implemented interpreter, when presented with this expression, looks up the symbol `id`, evaluate the symbol, which resolves to a function call to `(lit prim id)` with args of `(5 5)`.

In this case, as we are comparing `Numbers`, `id` works under the hood by comparing the digit values of the Python `ints` or `floats` used to construct the `Number`. If it is a pair, it uses Python's `is` to ensure they are both the same object in memory.

3.5.4 Special Forms

Special forms, as the name suggests, are alternate forms for expressions that are evaluated according to a different set of rules. We specify these rules in our evaluator. `quote`, for example, is a special form that tells the parser to represent the argument of quote as a Bel value. Quote is represented in Bel as `(quote a)`, or more commonly, `'a`.

Quote

An expression that conforms to the above specification should be passed to `special_quote`, which is a special form reached when the `car` of the expression is the Symbol `quote`, or `'`.

We can only quote one object at once with the quote operator. Evaluating an object that has been quoted removes the quote.

If

`if` operates as such in Bel. Consider the test below, which reads: if `2` is equal to `0`, then return `5`. Else, return `20`.

```
(if (= 2 0) 5 20)
```

We call this special form when we have the Symbol `if` in the first half of an expression. This is a pattern we'll see for most of the special forms. It is called on the tail of that expression and the lexical environment. `special_if` operates recursively, with a tail call on the then branch. Our `if` special form is implemented with tail call optimization, as we shall explain later.

If we have called `special_if` on a pair with a `cdr` of `nil`, then evaluate the first half of that pair. (In practice, `(if 5)` will then evaluate to `5`).

Else, if the `cdr` is not a pair, throw an error. Otherwise, it is a pair, and we can define the head of that pair as the test, and the head of the tail of that pair as the consequent. If we pass the test (the test evaluates to anything other than nil), then we evaluate the consequent and return that expression.

Otherwise, do a tail call on the tail, of the tail of the expression. This tail is either the final else branch, or another nested if expression. If it is the final else branch, then return the evaluation of that. Otherwise, call `special_if` recursively on the (implicit) nested if expression.

Define

`define` allows us to register new variable-value pairs in our global environment.

3.5.5 Function Application and Closures

`function_apply` is the application function. It takes a function and applies that to the list of evaluated arguments.

A function can be a primitive but can also be a closure.

In a closure, we bind arguments to the formal parameters, create an extended environment, and evaluate under the new environment.

In the case of function application, however, we have to be careful not to hit the maximum recursion depth. The challenge is that whenever we implement a recursive function in direct style (it calls itself in the body, which generates another pending result on the call stack), we run the risk of having the program crash due to consuming too much memory.

3.6 Tail Recursion is its Own Reward

Having implemented `eval`, we were hit with the challenge of tail-call optimization.

We see that a call to `eval`, like `function_apply`, generates subproblems for us to solve. This is similar for any function that calls itself recursively. At some point, if the programs are correct, then they will return a result to the function that called that function. Until then, however, it consumes space on the call stack, which could lead to a stack overflow. To understand how to solve this, we first must understand a grammar of Babybel expressions.

3.6.1 A Grammar of Expressions

Let us speak more explicitly of a grammar for Babybel expressions. Any correct Babybel expression will conform to this grammar, and can be evaluated by the evaluator, modulo any bugs.

```
e := <atom> | (if e e e) | (e {e}*) |
(fn <formals> e_tp) | (let ({(formal e)}*) e)


e_tp := <atom> | (if e e_tp e_tp) | (e {e}*) |
(fn <formals> e_tp) | (let ({(formal e)}*) e_tp)
```

A Babybel expression (`e`) is either an atom, or an if special form, a function, application, or a definition. A Babybel expression in tail-position (`e_tp`) is either an atom, an if special form, a function, application, or a definition.

A function application in tail-position is also known as a tail call [5].

For the evaluator to be properly tail-recursive, evaluating all `e_tp`'s must be carried out in a properly tail recursive way. This means that evaluation will not generate an additional return to the call stack.

We choose to implement a tail recursive version of our functions using a while loop. Therefore, we develop an `eval_tp` function that takes an `e_tp` and an environment as an argument.

Let us examine eval_tp, which at first glance, seems to do the same thing as eval, except wrapped in a while-loop. As most of the code is similar to eval, we will only include sections of particular interest here.

Its body will be within a while-loop, such that expressions that generate tail calls will resolve within the body of eval_tp. Witness eval_tp below:

```
def eval_tp(exp, l_env):
    while(True):
        ... if idp(exp.a, Symbol("if")): # special if
                body = exp.d
                if nilp(body.d):
                    exp = body.a
                    continue
                if not pairp(body.d):
                    raise SyntaxError("The cdr of this pair
                    has to be a pair.")
                else:
                    test = body.a
                    conseq = body.d.a
                    if not nilp(eval(test, l_env)):
                        exp = conseq
                        continue
                    else:
                        exp = Pair(Symbol("if"), body.d.d)
                        continue
        ... else: # function application
```

```
            fun = eval(exp.a, l_env)
            args = eval_list(exp.d, l_env)
            if primitivep(fun):
                return apply_primitive(
                    fun.d.d.a, args)
            elif closurep(fun):
                l_env_prime = fun.d.d.a
                params = fun.d.d.d.a
                l_env = Env(bind(params, args, \
                l_env_prime))
                exp = fun.d.d.d.d.a
                continue
        ...
```

3.6.2 If: revisited with a tail call

When we evaluate the body of an "if" expression, `special_if` is not called directly - we instead call `eval_tp` on that expression, which will hammer down the expression until we either get an error, or receive our desired result. This is done by assigning the test and consequent in an iterative fashion until we exit the loop. Assuming an `if` expression that conforms to our grammar, our exit conditions are as such: either we have hit the else case, where the `cdr` of the body is `nil`, or the test has passed – that is, evaluating the test with the lexical environment returns a non-nil value, in which case we assign the expression to the consqeuent and evaluate it. Otherwise, we continue iteratively through the remaining conditional branches.

3.6.3 Closures and tail recursion

In `eval_tp`, a tail-call is performed in a properly tail-recursive way by continuing the while loop on the body of the closure that is being tail-called.

3.7 Conclusion

We now have a properly tail-recursive Babybel evaluator that obeys left-to-right call-by-value, and we can use it to analyze programs of interest.

Chapter 4

Analyzing Bel Programs

4.1 Overview

This chapter will detail our efforts at using Babybel to analyze programs of interest. We discuss Church Numbers, Fibonacci and Y Combinators.

4.1.1 On Testing

Having written the unit tests for various functions, we can use the interpreter to evaluate these various unit tests and verify that our interpreter performs as expected.

4.2 Examples

4.2.1 Church Encodings

We utilize our bootstrapped Bel interpreter, implemented in Python, Babybel, to build up Church numbers using the standard functionality of Bel. This exercise will allow us to demonstrate the functionality of our environment and our `eval` function.

Church encoded natural numbers are a representation of the natural numbers using lambda calculus.

A church encoded natural number is recursively defined as such: it is either the number `0`, or it is the successor of another church encoded natural number.

Church encodings are implemented and demonstrated with these functions: `cn_zero`, which represents the natural number 0, `cn_succ`, which, given a natural number, returns its successor, `n2cn`, which takes a natural number and returns its church encoded form, and `cn2n`, which takes a church encoded number and returns its form as as our Bel Type, `Number`.

Once we have these Bel functions implemented, we can implement functions such as addition on Church numbers, multiplication on Church numbers, and exponentiation on Church numbers. These functions are implemented, and their behaviour is discussed in Appendix A.

Exponentiation on Church numbers proved particularly challenging as a test for our interpreter, as the large call stack generated by the function quickly caused a stack overflow. This required us to design and implement a properly tail-recursive version of `eval`, as explained in the previous chapter.

4.2.2 Infinite Self-References

Consider self-applying the function `(fn (x) (x x))`, which the cognoscenti will have recognized as the ω term in the λ calculus.

Evaluating this self-application diverges in that there is no stopping condition for the while loop that performs its successive tail calls.

In the initial test, our Wizard of Oz Babybel did not loop forever, but instead began sputtering from a stack overflow. This revealed that it was truly Python underneath, as it announced a recursion error, claiming that the `maximum recursion depth was exceeded."`

In order to pass this unit test, and as described in Chapter 3, we implemented an iterative version of `eval`, such that the response was not an immediate call stack overflow, but the desired behaviour of defining itself in an infinite loop.

4.2.3 Reversal of Lists

The reverse function, or `rev` helped us stress test our functions on lists, such as `car`, `cdr` and `cons`. We have also implemented a tail recursive implementation of reverse with an accumulator using Babybel.

```
(define rev_acc
  (fn (vs a)
    (if (pair? vs)
        (rev_acc (cdr vs) (cons (car vs) a))
        a)))


(define rev
  (fn (vs)
    (rev_acc vs nil)))
```

Calling `eval` on `(rev (cons 1 (cons 2 (cons 3 nil))))`, i.e, applying our reverse function to the list `(1 2 3)`, returns the Bel list `(3 2 1)`.

4.2.4 Fibonacci

The Fibonacci function helped us stress test calls and tail calls. We have implemented both the standard version and a linear version that enumerates consecutive Fibonacci numbers, as well as a version implemented in continuation-passing style (CPS) [5]. Since Babybel is properly tail recursive, CPS programs execute without any hiccup.

4.2.5 Y-Combinators

We have also experimented with Y Combinators, or fixed-point combinators [8], to encode recursion by self-application, in the manner of the ω term mentioned in Section 4.2.2. With a Y Combinator, we can implement recursive functions by lambda-abstracting their name, no matter their arity.

4.2.6 Variadic Functions

For convenience, we also implemented variadic functions. For example, the list function is defined as follows:

```
(define list
  (fn xs xs))
```

So now there is no need to write, e.g., `(cons 1 (cons 2 (cons 3 nil)))`. Instead, we simply write `(list 1 2 3)`.

Chapter 5

Discussions and Reflections

5.1 Reflections

It has been a challenge to understand and then implement (1) a reader and parser, and (2) an interpreter. We would like to extend our gratitude to (1) Professor Olivier Danvy and (2) the Lisp community, which continues to generously share their knowledge and findings.

In particular, we are grateful to Peter Norvig, [16] William Annis, [1] and Lucas Vieira [24] for their illuminating examples, and of course to Paul Graham for sharing his specification of Bel.

We implemented a properly tail-recursive interpreter for Babybel, and we tested it out on a range of programs, including list processing of course, but also Church encodings, CPS, and Y-Combinators.

In sum, we defined the grammar of the language, we teased out the specification, and developed the program with the specification in mind in a new meta-language, Python, which to the best of our knowledge had never been used before to implement Bel.

Throughout this project, I developed a finer understanding of Lisp, Lisp and hacker culture, the different tribes within Lisp, and how Lisp has impacted the development of programming languages, up until to-day.

This project has enriched me as a computer scientist and programmer in a substantive way. As for the next step, namely the famed profound enlightenment experience that will make me a better programmer for the rest of my days, I believe I am ready for it, in that I can now start to program in Lisp.

5.1.1 Alternative Approaches

Perhaps we could have considered a different implementation language. Why Python? It's the language I'm most comfortable with, and interestingly enough, this experience building the interpreter has taught me about the features I wish I had in Python (type annotations, proper forward declarations, tail-call optimization), and reminded me of the features of Python which I loved (namely its simplicity and its straightforward error reporting).

I chose to wrote this interpreter in Python because I would accomplish more than if I would have used a different tool. I'd be spending more time learning the language features and idioms.

We privilege the development of understanding over struggling with the "What" and "How" – I understand now Paul Graham's famous quip on Blub and how programming languages vary in power [9], but this power is relative to the strength of the wielder. A wizard cannot do much with a broadsword, until they train to wield it.

As Graham observed, programmers get "very attached to their favorite language". Paul Graham claims that Lisp is the most powerful language: because Lisp code is made of data structures one can traverse, one can manipulate programs and write programs that write programs. Whilst I didn't get a chance to immerse myself in Lisp development, I got a glimpse behind the curtain.

Because programmers think and program in their given language, in the same way that language speakers think in their native tongue, language paradigms change very slowly. Learning Lisp makes one more conscious of the power continuum and spectrum that exists for programming languages.

As for other evidence Lisp's continuing relevance in the 21st century, Clojure may be a good signpost towards how it's being used in production [13]. Every user of Emacs, a popular text editor, also is a Lisp user, whether they know it or not [18]. However, we have chosen to focus on Paul Graham's Bel, and those may be worthwhile topics for another paper.

When it comes down to it, this report is about implementing Babybel, in Python, and elucidating broader Lisp culture.

5.2 On Education

My claim after having completed this capstone thesis is that although Lisp may have fallen out of favour as an introductory language with the phasing out of Sussman's and Abelson's Structure and Interpretation of Computer Programs (affectionately known as the Wizard book), Lisp could still prove a powerful educational tool for computer scientists – if only to demonstrate the relative malleability of code and data, and how these are manipulated at a symbolic level.

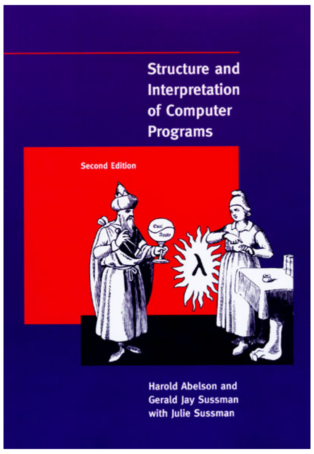

Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

FIGURE 5.1: The cover of Structure and Interpretation of Computer Programs, by Sussman and Abelson. Note the wizard on the left with the orb, and the yin-yang version of eval and apply. Note the matronly figure exclaiming "lambda" like a magical spell... and the table with a human foot? All this contributed to a mythos around Lisp.

To understand how interpreters and compilers are developed is truly a journey to understand computation, for one is forced to think how a computer parses the very words one is typing in order to obtain one's desired result.

In the process of this capstone, I have been encouraged to

1. reflect more on the development of computer science as a discipline over the last 50 years and become more thankful for some of the founding persons in computer science,

2. consider some of the origins and debates in computer science and their relevance today, particularly those that animated the development of Lisp in the 1980s and 1990s,

3. develop my conversant ability in both the meta-language (Python) and the implemented language (Babybel).

Failing anything else, making an earnest effort to learn Lisp will naturally encourage young computer scientists to think about recursive definitions of data structures such as lists as means to manipulate data.

5.3 Further Work

One can possibly extend this work to improve error-handling and optimize various functions, as well as implement streams, macros and all of those other juicy language features that Bel promises. We recognize that an interpreter is an ongoing project – if Paul Valéry programmed, he might have said that "an interpreter is never finished, only abandoned".

Error handling, first of all, could be incorporated within the Babybel REPL. As of now, the interpreter has implemented a minimal subset of Bel, and relies on Python for error handling.

The most exciting future work of all would look at extending the current Babybel implementation to successfully run a Bel self-interpreter, which Paul Graham has graciously provided the source for [10]. This would involve engaging in battle with macros. Then, we've truly begun with turtles.

Unlike some of the other projects and programs I've worked on, an implementation of a language is a far more difficult challenge, as the sky is the limit. One must recognize that at some point, *Worse is Better* [7] and that it may be best of all to send out what one has accomplished thus far into the world sooner rather than later.

Bibliography

[1] William Annis. *PyLisp - A Tiny Lisp in Python*. PyLisp - A Tiny Lisp interpreter in Python. Accessed: 28th March 2021. 2002 (cit. on pp. 9, 10, 36).

[2] ANSI. *ANSI INCITS 226-1994 (S20018)*. ANSI, 1994 (cit. on p. 4).

[3] Sean Champ. *Lost Ina Seaof Parentheses*. Accessed: 4th April 2020. 2001 (cit. on p. 3).

[4] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941 (cit. on p. 1).

[5] Olivier Danvy. *An Analytical Approach to Programs as Data Objects*. Aarhus, Denmark: AU Library Scholarly Publishing Services, 2006 (cit. on pp. 28, 35).

[6] Olivier Danvy. *Environments*. Accessed: 4th April 2021. 2021 (cit. on p. 15).

[7] Richard P. Gabriel. *Worse Is Better*. Accessed: 2nd April 2021. 1991 (cit. on pp. 12, 40).

[8] Mayer Goldberg. "A Variadic Extension of Curry's Fixed-Point Combinator". In: *Higher-Order and Symbolic Computation* 18.3/4 (2005), pp. 371–388 (cit. on p. 35).

[9] Paul Graham. *Beating the Averages*. Beating the Averages. Accessed: 4th April 2020. 2009 (cit. on pp. 3, 5, 37).

[10] Paul Graham. *Bel*. Bel. Accessed: 13th November 2020. 2019 (cit. on pp. 5, 14, 40).

[11] Paul Graham. *The Bel Language*. Paul Graham: The Bel Language. Accessed: 13th November 2020. 2019 (cit. on pp. 5–7).

[12] Paul Graham. *What Made Lisp Different*. Paul Graham: What Made Lisp Different. Accessed: 13th November 2020. 2001 (cit. on p. 2).

[13] Rich Hickey. *Clojure*. clojure.org. Accessed: 4th April 2020. 2021 (cit. on p. 38).

[14] John McCarthy. "History of LISP". In: *History of Programming Languages*. 1978, pp. 173–185 (cit. on p. 2).

[15] John McCarthy. "Recursive Functions of Symbolic Expressions And Their Computation By Machine, Part I". In: *Communications of the ACM* 3.4 (1960), pp. 184–195 (cit. on pp. 1, 6).

[16] Peter Norvig. *(How to Write a (Lisp) Interpreter (in Python))*. (How to Write a (Lisp) Interpreter (in Python)). Accessed: 2nd April 2021. 2010 (cit. on p. 36).

[17] Eric Steven Raymond. *How to be a Hacker*. Eric Steven Raymond: How to be a Hacker. Accessed: 13th November 2020. 2001 (cit. on p. 4).

[18] Richard Stallman. *My Lisp Experiences and the Development of GNU Emacs*. My Lisp Experiences and the Development of GNU Emacs. Accessed: 4th April 2020. 2002 (cit. on p. 38).

[19] *State of Common Lisp Survey 2020*. Accessed: 28th March 2021 (cit. on p. 4).

[20] Guy L. Steele Jr. and Richard P. Gabriel. “The Evolution of Lisp”. In: *History of Programming Languages Conference*. Ed. by Jean E. Sammet. ACM SIGPLAN Notices, Vol. 28, No. 3. Cambridge, Massachusetts: ACM Press, Mar. 1993, pp. 231–270 (cit. on p. 3).

[21] Gerald J. Sussman and Guy L. Steele Jr. *Scheme: An Interpreter for Extended Lambda Calculus*. AI Memo 349. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998, with a foreword. Cambridge, Massachusetts: AI Lab, MIT, Dec. 1975 (cit. on p. 3).

[22] *The SICP Website (MIT)*. The SICP Website. Accessed: 13th November 2020. 2021 (cit. on p. 3).

[23] Alan Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42.2 (1936-37). Corrections in Volume 43, pages 544-546, 1937, pp. 230–265 (cit. on p. 6).

[24] Lucas Vieira. *Believe: A Bel Lisp Interpreter Built with C*. Believe: A Bel Lisp Interpreter Built with C. Accessed: 2nd April 2021. 2020 (cit. on p. 36).

[25] YCombinator. *Y Combinator Top Companies*. Y Combinator Top Companies. Accessed: 13th November 2020. 2020 (cit. on p. 5).

Appendix A

Babybel Source Code

We have provided the complete source code of Babybel, a scoped-down implementation of Bel in Python, in a live GitHub repository.

We instruct the user how to initiate an instance of the Babybel Read-Eval-Print-Loop (REPL) that will allow them to test Bel programs, and have included sample test programs.