Macro-driven metalanguage for writing Pyramid Schemes programs

Marcin Konowalczyk[1, 2, a)]

[1)] *Department of Chemistry, University of Oxford, Chemistry Research Laboratory, Oxford OX1 3TA, U.K.*

[2)] *UCLA Samueli School of Engineering, University of California, Los Angeles, 7400 Boelter Hall, Los Angeles, CA 90095, United States*

(Dated: March 26, 2021)

In this work we present a metalanguage which allows simpler writing of Pyramid Scheme programs. We first introduce the Pyramid Scheme itself, pointing out some more interesting features. We then proceed to define a base lisp-like notation for Pyramid Scheme (called `psll`), and expand on it with local macros (and semi-local) macro expansions which allow for higher-level constructs. Notably, we introduce strings, arrays and preprocessor definitions which can be used akin to functions. The entire project is available on GitHub at MarcinKonowalczyk/psll-lang.

Keywords: syntax tree; pyramids; compilation; horizontal gene transfer; sorting; code golf

I. INTRODUCTION

In ancient Egypt, pyramids were constructed as the resting places of deceased pharaohs, containing not only their mummified remains but also an assortment of keywords and type literals the pharaoh will need in their journey though afterlife. Pyramid Scheme (PS) is a variant of the Scheme dialect of Lisp, which honours these ancient traditions and accompanies *us* thorough our journey of computation.

Pyramid Scheme was designed by Conor O'Brien, in early 2017 (date of the earliest commit to the GitHub repository).[1] It is a turing-complete esoteric programming language (esolang)[2,3] which uses tree-like, as opposed to a serial, code structure. Compilers make use of an intermediate representation of the language in the form of an abstract syntax tree (AST).[4] In contrast to most contemporary languages / frameworks, which build on top of the existing infrastructure to create "the stack" of software,[5,6] Pyramid Scheme aims to shed any unnecessary abstractions, including that of the AST. The computation in Pyramid scheme is therefore represented as a literal syntax tree (LST) of ascii-art pyramidal constructs.

Pyramid Scheme is supported by the "Try It Online!" repository of online interpreters,[7] and, like many other esolangs, has been featured in many code golfing challenges.[8] Code golf involves writing a program in a freely-chosen programming language which performs a certain operation under some constraint. This usually comes in the form of the smallest number of characters in the source code and is a platform for one to either learn a new programming language, or explore the depths of an already known one. Code golfing provides one with a set of goals which is almost-orthogonal to what one finds in everyday programming, and therefore often sheds new light on old, seemingly well-known ideas.

```
      ^ ^     ^             ^     ^     ^    ^
     ^-/ \   / \           / \   / \   / \  -^
    ^-/out\ /set\         /out\ /out\ /out\  / \
   / \-----^-----^       ^----- ^----- ^----- /out\
  /set\   /x\   /+\     / \   / \   / \    ^-----
 ^-----^  ---  ^---^   /chr\ /chr\ /chr\  /x\
/x\   /#\     /x\ /1\ ^----- ^----- ^-----  ---
---  ^---     --- ---/ \   / \   / \
    /1\           /43 \ /49 \ /61 \
   /ine\          -----  -----  -----
   -----
```

Listing 1. A simple Pyramid Scheme program. It takes one input from `stdin` – `(set x (# stdin))`, increments it by one – `(set x (+ x 1))` and prints the result computation to the command line. Try it online!

II. PYRAMID SCHEME

The original and, so far, the only implementation of PS is written in Ruby.[1] The LST of the program is first parsed and then mapped to a recursive evaluation chain. An example of one such program can be seen in Listing 1.

PS parser reads the body of each pyramid verbatim, concatenated line by line.[9] The parser begins at the tip (`^`), and walks down the left (`/`) and the right (`\`) side, collecting the characters in-between. When the two sides run out, it first checks for the presence of the pyramid base (`-`),[10] and then for the tips of the child pyramids, if present. The pyramids may connect *only* on these corners, such that, for example, the first pyramid with `chr` (which constructs a character `+` to be printed) in Listing 1 rightfully does not consider the pyramid `1` of the `set` branch as its child.

Note, however, that this allows for an existence of direct con-

[a)] Electronic mail: marcin.konow@lczyk.xyz

nection between neighbouring branches of the LST – in Listing 1, for example, the first print statement (`out` keyword), shares the node `x` with its neighbouring branch. This is an interesting parallel to the phenomenon of the lateral gene transfer observed in genetics, and suggests a more-proper description of the PS to be that of a Ewok village syntax tree (EVST).[11,12] Although this is undoubtedly one of the more interesting and powerful features of PS, it has not yet been implemented in the project described herein shortly, and therefore will not be considered further, but left for future work.

The specification of the pyramid structure does not preclude the existence of a pyramid with no content. Such a 0-height pyramid is falsey and evaluates to 0.[13,14] A pyramid with no content *does* however both evaluate its children, and pass them as an its output. This make the 0-height pyramid an important construct for code packing, as can be seen in the first branch in Listing 1

There are two types operators in PS: ones which implicitly evaluate both of their children, as well as those which do this only under certain circumstances. The first group maps very closely to its underlying Ruby implementation. There are basic binary arithmetic and comparison operators: `+`, `*`, `-`, `/`, `^`, `=`, `!` and `<=>`. Keyword `out` prints all of its inputs and `chr` converts number to a character. The keyword `arg` indexes arrays (or input arguments), and keywords `#` and `"` convert back and forth from and to a string. `#` character also allows one prompt user for input if given a (semi)keyword `line`.[15,1]

The second group of operators conditionally evaluates only one of their children. `set` sets the variable denoted by its left child to the evaluated right one. `loop` and `do` evaluate the right child subtree as long as the left one is truthy (with the difference being when is the check made – before and after right subtree evaluation respectively). Finally, keyword `?` evaluates the right subtree only if the left one truthy, else it evaluates to zero.

III. PSLL

In order to assist the programmer in harnessing the power of Pyramid Scheme, we introduce a meta-language - Pyramid Scheme lisp-like notation (`psll`).

a. Bracket structure Lets consider the LST approximation of the full EVST structure of Pyramid Scheme. Every node of the LST consists of at most three pyramids - a parent and two children, maybe. A node will, therefore, be represented by a bracket containing exactly three, space-separated words, brackets or null-markers (`_`). Only the first entry is allowed to be a word. A simple statement in such notation may be `(set (x _ _) (+ (x _ _) (1 _ _))` – the second branch from Listing 1, increment variable `x` by one. Although this is sufficient to re-serialize any PS program, one quickly notes the cumbersomeness of having to specify the empty space explicitly. Therefore we add a simple macro-like expansion where, firstly, each lone word in the 2nd or 3rd position is considered to be in a bracket of its own, and secondly each bracket with length of less than 3 is expanded up to the length of 3. Hence the increment branch can be written as `(set x (+ x 1))`, since `x → (x) → (x _ _)`. This also means that keywords with less than two arguments do not need to specify explicit null-markers for the second argument. Lastly `//` denotes a comment. Hence, the program from Listing 1 can be written as:

```
(set x (# line)) // Get x from stdin
(out _ x) // Print x
(set x (+ x 1)) // Increment x
// Print "+1=" and then the value of x again
(out (chr 43)) (out (chr 49)) (out (chr 61)) (out x)
```

Listing 2. LST approximation of the program from Listing 1 in simple `psll` notation.

Note that the LST approximation has been applied, such that `x` from `out` and `set` are now different. To get the code in Listing 1 the PS source has been modified by hand post compilation.

This type of local macro (compile-time code alteration) expansion is at the core of `psll`. Such macros do not add any expressive power to the language,[16] but allow one to use higher-level constructs and simplify writing programs. All of the functionality, which will be described shortly, has been implemented by repeatedly leveraging a single python function which performs a depth-first walk through the AST and applies functions at the appropriate nodes (Listing 3).

```
def tree_traversal(ast, pre_fun=None, str_fun=None,
    post_fun=None, final_fun=None):
    ast2 = [] # Since, ast is immutable, build a new ast
    for node in ast:
        if node is None:
            ast2.append(node)
        elif is_string(node):
            ast2.append(str_fun(node) if str_fun else node)
        elif is_tuple(node):
            node = pre_fun(node) if pre_fun else node
            node = tree_traversal(node, pre_fun, str_fun,
                post_fun, final_fun)
            node = post_fun(node) if post_fun else node
            ast2.append(node)
        else:
            raise TypeError
    ast2 = tuple(ast2)
    final_fun(ast2) if final_fun else None
    return ast2 # Return ast back as a tuple
```

Listing 3. Core `psll` function performing a depth-first walk through the abstract syntax tree and application of appropriate functions.

b. Implicit bracket expansion Each bracket must have exactly three elements. For small expressions this is almost always the case, but becomes problematic for larger, flow-control and loop structures where each such expression can contain an arbitrarily large number of sub-expressions

which would then have to be manually nested in empty subtrees. An overfull bracket is one containing more than two other brackets, such as:

```
( (out 1) (out 2) (out 3) (out 4) (out 5) )
```

gets expanded as:

```
( (((out 1) (out 2)) ((out 3) (out 4))) (out 5) )
```

Each neighbouring pair of elements of the parent gets put together into a bracket, until the length of the parent is less than 2. Then, each bracket with exactly 2 other brackets has the empty-marker inserted as the first element. Note that the empty marker is a compiler-only keyword (python empty string) and it cannot be typed directly.[17] This results in a (literal) balanced binary tree in the final PS code, and so for a parent bracket of N sub-expressions will result in a tree of containing $\mathcal{O}\left(\log_2(N)\right)$ pyramids.

c. Expansion of binary operators A similar type of expansion can be applied to a bracket where the first member is not a child bracket but a keyword. This is done only for all binary operator keywords (+, * as well as -, /, ^, = and <=>) in a left-associative (LA) fashion, such that:

```
(+ 1 2 3 4) // This
(out (+ (+ (+ 1 2) 3) 4) newline) // Becomes this
```

Addition and multiplication are commutative over the set of most possible inputs, and hence the exact order of operations does not usually matter (string multiplication overloads concatenation and that's not commutative). For a non-commutative operation, e.g. subtraction, the expansion order does matter. Hence, if the keyword is placed at the end of the bracket, a right-associative (RA) expansion is performed:

```
(- 1 2 3 4) // This
(- (- (- 1 2) 3) 4) // Does indeed expand into this
(1 2 3 4 -) // But this
(- 1 (- 2 (- 4 3))) // Expands to this instead
```

Note that the order of the last two elements is purposefully reversed, such that the RA expansion is symmetrical with respect to the LA one. For the sake of compatibility with non-expanded brackets, the following two are also allowed for all binary operators.

```
(- 1 2) // eval to -1
(1 2 -) // eval to +1 since arguments reversed
```

Finally, the `out` keyword normally does not allow for output of more than 2 variables. In `psll` the `out` keyword can have any number of inputs, and it gets implicitly expanded to a chain of output statements:

```
(out a b c d e) // This
(out a b) (out c d) (out e) // Becomes this
```

Note that this is different to the left-associative expansion of the binary keywords above. There is no right-associative expansion of the `out` keyword.

d. String literals Single characters can be created in the Pyramid Scheme memory with the `chr` keyword (Ruby `.to_i.chr`). It is also possible to construct longer strings since Ruby's "+" sign overloads string concatenation. The string `hello` is therefore:

```
(+ (chr 72) (chr 101) (chr 108) (chr 108) (chr 111))
```

Where the numbers are the decimal ascii codes for the respective letters, and a LA + operator expansion has been assumed. `psll` introduces string literals, such that `"hello"` expands into the above code.[18] The simplest "Hello, Sailor!" program in `psll` is `(out "Hello, Sailor!")`.

e. Array literals Arrays are created in Pyramid Scheme when an empty node has two subtrees. The subtrees get evaluated and concatenated into a length-2 array.[19] Repeated evaluation through nested trees doesn't produce longer but nested arrays. Ruby's + operator overloads array concatenation and allows one to create longer arrays.

```
(set a (1 2)) // Length-2 array
// This results in nested arrays
(set a (3 (1 2)))
(set a ((1 2) 3))
// Add arrays to make longer ones
(set a (+ (1 2) (3 4)))
```

This approach is, however, not fully general, as it does not allow for creation of odd-length arrays, nor an empty array. These can be made since Ruby's - overloads array difference (filtering):

```
(set a (- (0 1) (1 1))) // Length-1 arrays
(set a (- (1 1) (1 1))) // Empty array
```

An array of any length can be made this way. `psll` array literals are denoted with square braces. Due to the order of literal expansion, they can contain string literals, as well as numbers and floats and variable references.

```
(set a 7)
(set b [1 "hello" "sailor" 3.1415 2 b 3 "[" ")"])
```

Note that no escape characters are needed for the brace characters in strings. The context manager is a particularly tricky part of the parser. To reduce it's complexity, brackets are not allowed inside of arrays. If they were, one could create nested environments (array in bracket in array in bracket etc.) which would have to be recursively parsed. The current version of the context parser (`context_split`) is non-recursive and linear in the size of the input.

Only one additional array keyword is currently implemented:

```
(set a (range 1 5)) // This
// Expands to this
(set a
    (+ (1 2) // Array [1,2]
        (+ (3 4) // Array [3,4]
            (- (5 1) (1 1)) // Array [5]
        )
    )
)
```

Note that `psll` is insensitive to indentation, and it has been used here purely to aid readability.

Keyword `range` can also create ranges with different step size, but cannot create ranges for variables, since the expansion is happening at compile time:

```
(range 0 10 3) // [0,3,6,9]
(set a 10) (range 0 a 3) // Fails
```

f. Definitions Compile-time definitions and their expansion are, so far, the only semi-local macro. Any `(def name ...)` construct gets replaced by a stub tree – `()` and corresponding definition is stored on a stack. Any string gets matched against `name`s in the stack, top down, and is replaced by the first match (or not at all). Upon leaving the bracket (the scope of the `def`), the stack is popped a number of times equal to the number of stub trees in the scope which is being left.[20] This is, in fact, the use of `final_fun` in Listing 3. All the defs are stored on the stack fully expanded, such that they can be used in other defs downscope. Since defs are parsed and their replacements are made on a single tree traversal, the order of the definitions matter and they cannot be used before they get defined, even within a scope.

```
(set a 0) (set b 0)
(def incr (set a (+ a 1))) // Increment a
(incr) // a = 1, b = 0
( // Open new scope
  // Redefine incr to increment b
  (def incr (set b (+ b 1)))
  (incr) // a = 1, b = 1
)
// Back to the definition from before the scope
(incr) a = 2, b = 1
```

g. Optimisation Since one of the goals of `psll` is to allow one to write compact Pyramid Scheme programs (for the purposes of Code Golfing, Section I), it implements a few optimisation algorithms. The AST of the `psll` program is first passed through a processing stack of tree traversals implementing macros for all of the above features. This pre-processed AST is then passed to the optimisation stage. Greedy optimisation, for example, considers all the possible pairs of branches, as well as single branches of the root level LST and attempts to insert an additional empty tree around each such pair/singleton.[21] It immediately accepts the first candidate with a smaller number of characters in the compiled LST and repeats the entre process. It halts if the attempt of inserting the empty pyramid at any of the candidates does not produce a smaller LST.

Currently this is one of the only two, rather similar optimisation algorithms, the other differing slightly in the number of candidates it considers, as well as taking the `min` of each iteration, as opposed to greedily accepting the first better candidate. Both of these methods can result in large reduction in the codebase of elaborate Pyramid Schemes;[22] however they can only add pyramids and never remove or combine them. Empty pyramids cannot be removed arbitrarily since this could disrupt the evaluation order and break implicit parent-child relationships between parts of the LST. To perform this type of optimisation, the algorithm will have to understand, at least partially, the context within which it is operating – something which existing algorithms do not take into account. Another interesting direction for the optimisation would be to optimise different features of the LST, for example its width, height, or some arbitrary packing density heuristic.

Note that, regardless of the algorithm and the target of the optimisation, it is crucial that the final step of the compilation – conversion from `psll` AST to the Pyramid Scheme LST (i.e. the Pyramid Scheme source code) needs to be performant, as it will likely be happening thousands of times for any optimisation algorithm. Luckily this process has been made rather robust, and is filled with readily cacheable intermediate results (subtrees don't change much).[23]

h. Sharp edges Despite authors best efforts, the introduction of syntactic sugar into `psll` introduces some edge cases which one ought to watch out for. Some, which are considered bugs, have been mentioned already but there are some which are indispensable, since they interact with other features of the language. The underscore keyword (`_`) is one such example – it is rarely, if ever, used yet it carries with it syntactic meaning. This could lead to confusion.

The other sharp edge is due to the fact that `psll` re-used `"`, `[` and `]` symbols for its own purposes of string and array literals respectively. These are also Pyramid Scheme keywords and therefore, when typed in `psll` they have to be escaped with a backslash.

IV. SAMPLE PROGRAMS

Having introduced the `psll` language, let us see what can be done with it.

a. Linear congruential generator A simple (cryptographically insecure!) pseudorandom number sequence can be generated with a linear congruential generator (LCG). A *very simple* LCG starts with a seed value, a prime multiplier, and a modulo base. The value of the generator changes from one iteration to the next according to the formula:

$$V_{n+1} = \mathrm{mod}\left(p\,V_n, d\right)$$

where V_n is the value of the LCG at iteration n, p is the prime and d is the modulo base. To get the output to be in the range 0-1, one only has to divide V_n by d.

Since PS does not implement the modulo function, we have to write it ourselves. In this case we use a very simple implementation which repeatedly subtracts d from $p\,V_n$ until the result is smaller than d. A small prime factor has been chosen to minimise the runtime.

```
(set value 312312) // seed the lcg value
(set div (^ 2 16)) // 16-bit divisor / modulo base
(set prime 7) // Prime factor

// Uniformly distributed random number between 0-1
// mod(prime*value + current, 2^16)
(def roll (
    (set value (+ (* value prime) 1))
    (loop // mod(value,div) by repeated subtraction
        (<=> (<=> value div) -1)
        (set value (- value div))
    )
    (set rand (/ value div))
```

```
))

// Print 100 such numbers
(set i 0)
(do (<=> i 100) (
    (roll) (out rand "\n")
(set i (+ i 1))
))
```

Listing 4. Simple linear congruential pseudo-random number generator. Try it online!

When compiled and run, it steps the LCG 100 times and prints the resulting uniformly distributed random numbers. Here are the first 7:

```
0.3585357666015625
0.5097656250000000
0.5683746337890625
0.9786376953125000
0.8504791259765625
0.9533691406250000
0.6735992431640625
```

b. Bubble sort As the final flourish, here is an implementation of bubble sort in `psll`. Bubble sort goes through a list, compares each pair of elements and, if appropriate, swaps them to appear in ascending order. At the end of the scan, the algorithm runs again if any swaps ocurred or halts if none did. Bubble sort is far from an efficient sort, but it is straightforward to implement, and therefore has been chosen here.

```
(set n (arg 999)) // Make nil value

// Array to be sorted
(set a [3 1 4 1 5 9 2 6 5 3 5])

// Get array length
// This will be: (len a N)
(set N 0) // Pointer into the array
// Increment pointer until goes off the end
(loop (! (= (arg a N) n)) (set N (+ N 1)))

// Append element of a in position q to b
(def append (set b (+ b (- ((arg a q) n) (n n)))))
// Usage: (set q ...) (append)

// Bubble sort the array
(do again (
    (set again 0)
    (set p 0) // Position pointer
    (loop (! (! (<=> p (- N 1)))) ( // For all pairs
    (set this (arg a p))
    (set next (arg a (+ p 1)))
    // This and next need swapping
    (set swap (! (<=> (<=> this next) -1)))
    (? swap (
        (set again 1) // Will need to go again
        (set b []) // Start b as an empty array
        // Add prefix of a
        (set l 0)
        (loop (= (<=> l p) -1) (
            (set q l) (append)
        (set l (+ l 1))
        ))
        // Add two elements, swapped
        (set q (+ p 1)) (append)
        (set q (+ p 0)) (append)
        // Add suffix of a
        (set l (+ p 2))
        (loop (= (<=> l N) -1) (
            (set q l) (append)
        (set l (+ l 1))
        ))
        (set a b)
    ))
    (set p (+ p 1)) // Increment position pointer
    ))
    (out (* a ",") "\n") // Print a
))
(out "done")
```

Listing 5. Bubbble sort of an array in `psll`. For demonstration purposes the array has been hardcoded.

When compiled and run, it produces the following output:

```
3,4,1,5,9,2,6,5,3,5,1
4,3,5,9,2,6,5,3,5,1,1
4,5,9,3,6,5,3,5,2,1,1
5,9,4,6,5,3,5,3,2,1,1
9,5,6,5,4,5,3,3,2,1,1
9,6,5,5,5,4,3,3,2,1,1
9,6,5,5,5,4,3,3,2,1,1
done
```

The compiled LST can be seen in Listing 6 (in the appendix).

V. CONCLUSIONS AND OUTLOOK

"Program in Pyramid Scheme! Teach your friends! Have them teach their *friends! Then have those friends teach* their *friends! ..."*

This is by no means a done project, so long as it is a platform for learning and having fun. The future direction of `psll` poses some genuinely interesting computational problems, such as efficient optimisation algorithms and performing context-aware transformations on the AST. The language does not currently allow one to leverage the full power of EVSTs of Pyramid Scheme, but instead uses the LST approximation. The goal is, indeed, to add this to the the language. This will, however, be a major milestone since the EV structure of the resulting syntax tree will require restructuring of the internals of the compiler. At least initially, EV branching will be available only at the level of intermediate-representation optimisers. However, since one of the purposes of `psll` is an esoteric flavour of code-golf, one might want to manually adjust the code structure, similarly to how the underscore keyword is used at the moment. Additional keywords, as well as their supporting architecture, will need to be introduced to be able to explicitly specify EV cross-branching structure.

There are a few major parts of `psll` which need to be fin-

ished before that. Notably there are a few core bugs which any additional functionality would make only harder to track. These are detailed in `README` in the main `psll` repository and range from relatively harmless (`def` inserts an extra empty pyramid) to major (`()` unduly pops the definition stack). There are also some minor support keywords which are yet to be added. These are, for example, `len` – a concise form of line 10 in Listing 5 and `nil` – a concise form of `(set nil (arg 999))` in the preamble.[24] This is not to mention typical and necessary software project irks like ensuring the project has appropriate test coverage (currently at 69%) and fighting code bloat (currently at approx 530 core lines + bash support).

Interestingly, `psll` caters to a new flavour of code-golfing. Large PS programs are not feasible to be written by hand, not to even mention the number of rewrites and code obfuscation which usually happens when golfing. Hence, all the golfing happens at the level of writing compiler and optimisation algorithms therein, rather than the code itself.[22]

Finally, very programmer shares a certain latent interest in the underlying structure of the languages they use every day. We would encourage them to scratch that itch. There are plenty of resources to start, but we are inclined to mirror the advice of Casey Muratori:[25] *"Look at all of the resources on these topics in in the following way: rather than reading what someone tells you about how to build a compiler (...) start programming one without knowing what you're doing (...) and see what you can learn. When you cannot make forward progress (...) [look for] solution to that particular problem you're having. (...) Now you have some context to evaluate what people tell you (...) whereas if you read about stuff without ever actually having encountered a problem yet, then you're just gonna have no idea [whether its valuable]."* If one really wants a starting point though, David Beazley's ply and sly projects,[26–28] are a good place to do so. Tey are a python implementation of common parsing tools `lex` (Lexical Analyzer Generator) and `yacc` (Yet Another Compiler-Compiler).[29] Also, Jonathan Blow is streaming, and uploading recordings of their work on a programming language called `jai` which is currently under development.[30]

VERSION NOTES

At the time of writing, the commit SHA of the main Pyramid Scheme GitHub repo is:[1]

`fd183d296f08e0cba8bf55da907697eaf412f6a7`

and the `psll` repo:[31]

`fbd8966231875196536bba0dee4cc0a970164b6f`

The `psll` repository also has all the latex and make files for this very paper. Short of fixing typos, the text will not be modified after the submission.

`psll` has been written in python >3.6. The only non-core library it depends on is `more-itertools` version, at least, 8.5.0. This dependency was thought to be appropriate since this work led to a pull request to `more-itertools`, added in version 8.5.0.[32]

Pyramid Scheme is written in pure Ruby. At the time of witting it works in Ruby version 3.0.0p0 (2020-12-25 revision 95aff21468)

ACKNOWLEDGEMENTS

I would like to thank Dr Hugh Lindley and Blaine Rodgers for proof-reading and helpful comments on the manuscript, Samuel Hutton for helpful discussions, as well as Jonathan Blow and David Beazley, for sparking a long-lasting interest in programming languages.

Last but not least, I would also like to cordially thank *you* dear reader. You have made it! Thank you for reading!

REFERENCES

[1] Conor O'Brien. Pyramid Scheme. GitHub repository, https://github.com/ConorOBrien-Foxx/Pyramid-Scheme, 2017.
[2] Pyramid scheme. Esolang wiki, https://esolangs.org/wiki/Pyramid_Scheme.
[3] Blaine Rodgers. High-Octane Rumble Simulation Engine. GitHub repo, https://github.com/PaperclipBadger/high-octane-rumble-simulation-engine, 2017.
[4] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.
[5] Bryan Cantrill. Zebras All the Way Down. Uptime 2017, https://youtu.be/fE2KDzZaxvE.
[6] Casey Muratori. The Thirty Million Line Problem. https://youtu.be/kZRE7HIO3vk, 2018.
[7] Try It Online! https://tio.run.
[8] Code Golf Stackexchange. https://codegolf.stackexchange.com.
[9] Hence, for example, the bottom pyramid in the first stack in Listing 1 contains the (semi)keyword `line`, as opposed to two words: `l` and `ine`.
[10] Note that the base of the pyramid is a dash (`0x2d`), not an underscore.
[11] Patrick J. Keeling and Jeffrey D. Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 2008.
[12] Zachary Weinersmith. Ewok Village of Life. SMBC, https://www.smbc-comics.com/comic/2012-04-08.
[13] The term "0-height" can be ambiguous since the pyramid itself has height of 2 characters. In this work the pyramid's height, however, is the number of lines of the text in its body.
[14] Conor O'Brien. Pyramid Scheme Negation. https://codegolf.stackexchange.com/questions/147513/pyramid-scheme-negation.
[15] Words `line`, (as well as `stdin`, `readline`) are referenced to as *semi*keywords since they have a keyword meaning only when they're an input of the # command.
[16] Shriram Krishnamurthi. On the Expressive Power of Programming Languages. PWLConf, https://youtu.be/43XaZEn2aLc, 2019.
[17] For completeness' sake this will likely be implemented by reusing the underscore keyword, such that, for example, `((out 1) (out 2))` could be then explicitly expanded in `psll` as `(_ (_ out 1) (_ out 2))`.
[18] Note that this is a very left-child heavy tree. To balance it, the above string could also be made by recursively concatenating its binary split. This will be implemented in the future.
[19] The key is the `unwrap` function – body of each empty pyramid and in the PS compiler. It returns the array element if passed only one input, but the entire array if two (`t.size == 1 ? t[0] : t`).
[20] This way of keeping track of definitions does, currently, lead to a bug where a stub tree in `psll` source code causes a compilation fail since it unduly pops the definition stack. This issued will be addressed, possibly with a different way of keeping track of defs in the scope. This is not,

however, a trivial change as it requires the tree traversal function to retain state about each scope through each recursive call.

[21] This can be done at any parent-child connection in the ast since the resulting empty pyramid will evaluate its child and pass it to the parent in the same manner as is they had a direct connection. Scoping for definitions does not matter since the optimisation is performed *after* on the fully-expanded AST – after all the macros have been applied.

[22] Marcin Konowalczyk. Pyramid Scheme Negation in Pyramid Scheme. https://codegolf.stackexchange.com/a/208938/68200.

[23] This is, in fact, the reason why the AST is represented as an immutable data structure. Mutable data structures cannot be cached.

[24] `(set nil (arg 999))` is just a way of generating nil value in memory and assigning it to a variable called `nil`. Ideally a more robust solution will be found.

[25] Jonathan Blow and Casey Muratori. Q&A: Making Programming Language Parsers. https://youtu.be/lcF-HzlFYKE, Starting at minute 8.00, 2020.

[26] David Beazley. Reinventing the Parser Generator. Pycon 2018, https://youtu.be/zJ9z6Ge-vXs.

[27] David Beazley. SLY (Sly Lex-Yacc). GitHub repository, https://github.com/dabeaz/sly.

[28] David Beazley. PLY (Python Lex-Yacc). GitHub repository, https://github.com/dabeaz/ply.

[29] John Levine, Doug Brown, and Tony Mason. *lex & yacc*. O'Reilly Media, Inc., 2nd edition, 1992.

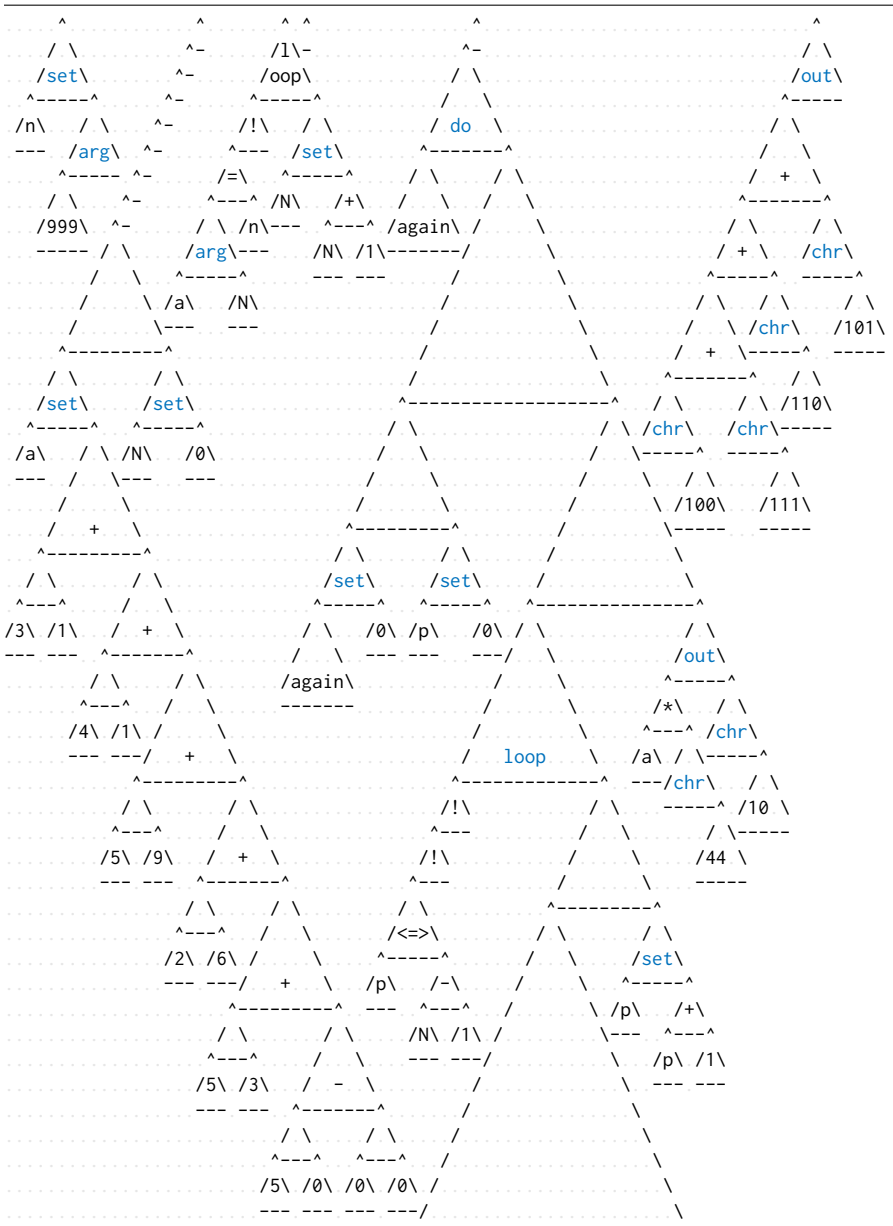
[30] Jonathan Blow and Casey Muratori. Making Programming Language Parsers. https://youtu.be/MnctEW1oL-E, 2020.

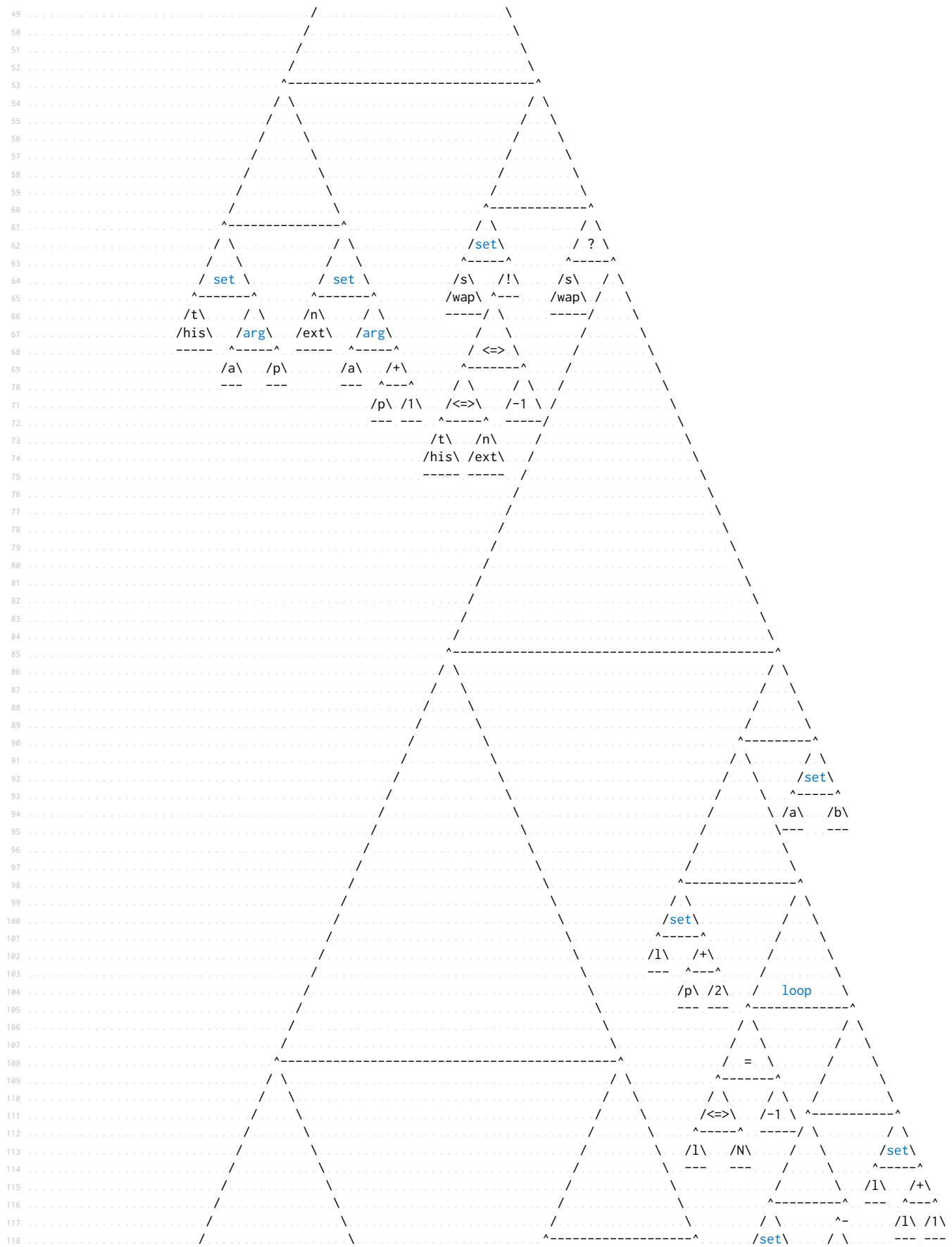
[31] Marcin Konowalczyk. psll-lang. GitHub repository, https://github.com/MarcinKonowalczyk/psll-lang, 2020.

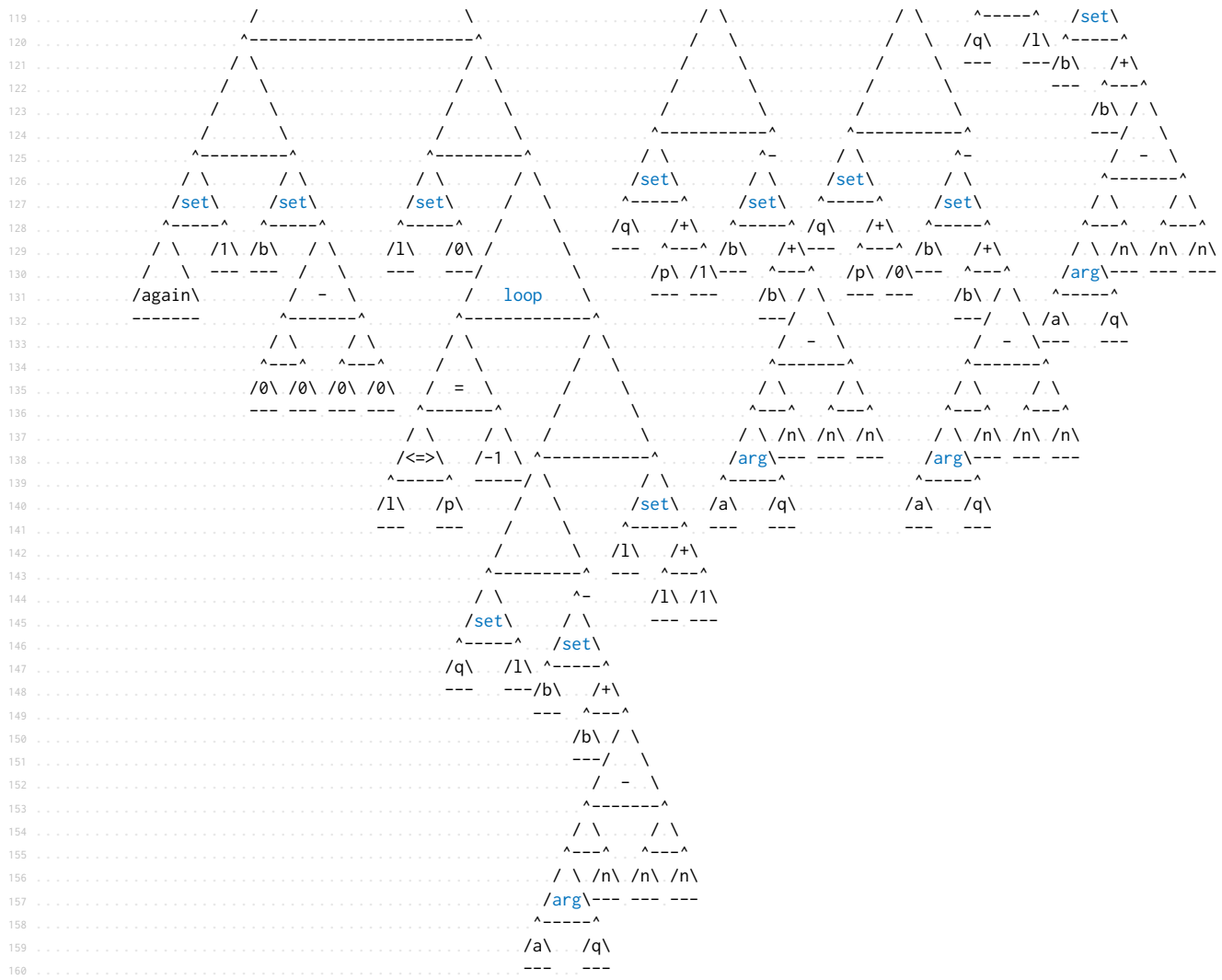
[32] Erik Rose and Bo Bayles. `more-itertools`. GitHub repo: https://github.com/more-itertools/more-itertools.

[33] The Lex & Yacc Page. http://dinosaur.compilertools.net.

APPENDIX




set
set
/t
/his
arg
ext
arg
/a
/p
/a
/+
/p
/1
/<=>
/-1
/t
/n
/his
/ext
set
/a
/b
set
/l
/+
/p
/2
loop
=
/l
/N
/set
/l
/+
/l
/1
/set



Listing 6. Bubble sort in Pyramid Scheme. Compiled with `-full-names` and `-co` (considerate optimisation) flags. The single letter `l` has been used instead of more verbose `nil` to reduce the width of the LST. Try it online!