

Local-macro-driven metalanguage as a platform for writing large Pyramid Scheme 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 ps11), 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](https://github.com/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 through afterlife. Pyramid Scheme (PS) is a variant of the Scheme dialect of Lisp, which honours these ancient traditions and accompanies *us* through our journey of computation.

PS was designed by Conor O'Brien, in the early 2017 (date of the earliest commit to the GitHub repository).¹ It is a turing-complete esoteric programming language (esolang)² 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).³ In contrast to most contemporary languages / frameworks, which build on top of the existing infrastructure to create “the stack” of software,^{4,5} 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,⁶ and, like many other esolangs, has been featured in many Code Golfing challenges.⁷

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.

```
1      ^ ^      ^      ^      ^      ^
2      ^- / \   / \      / \   / \   / \   ^-
3      ^- /out\ /set\      /out\ /out\ /out\ / \
4      / \-----^-----^-----^----- /out\
5      /set\  /x\  /+\      / \   / \   / \   ^-----
6      ^-----^  ^-----^  /chr\ /chr\ /chr\ /x\
7      /x\  /#\      /x\ /1\  ^-----^-----^-----
8      ^-----^  ^-----^  / \   / \   / \
9      /1\      /43\ /49\ /61\
10     /ine\      ^-----^
11     ^-----
```

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.

PS parser reads the body of each pyramid verbatim, concatenated line by line.⁸ 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 (-),⁹ 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 connection 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).^{10,11} 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

^{a)}Electronic mail: marcin.konow@lcyk.xyz

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 0.¹²¹³ 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.¹⁴¹

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 evaluates to true (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 evaluates to true, 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).

Bracket structure Lets consider the LST approximation of the full EVST structure of Pyramid Scheme. Every node of the LST has at most two children. We will express each node as 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 its 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 $x \rightarrow (x) \rightarrow (x _ _)$. 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:

```
1 (set x (# line)) // Get x from stdin
```

```
2 (out _ x) // Print x
3 (set x (+ x 1)) // Increment x
4 // Print "+1=" and then the value of x again
5 (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 andset 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 expansion is at the core of psll. Such macros do not add any expressive power to the language,¹⁵ 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).

```
1 def tree_traversal(ast, pre_fun=None, str_fun=None,
2 ... post_fun=None, final_fun=None):
3 ... ast2 = [] # Since, ast is immutable, build a new ast
4 ... for node in ast:
5 ...     if node is None:
6 ...         ast2.append(node)
7 ...     elif is_string(node):
8 ...         ast2.append(str_fun(node) if str_fun else node)
9 ...     elif is_tuple(node):
10 ...         node = pre_fun(node) if pre_fun else node
11 ...         node = tree_traversal(node, pre_fun, str_fun,
12 ... post_fun, final_fun)
13 ...         node = post_fun(node) if post_fun else node
14 ...         ast2.append(node)
15 ...     else:
16 ...         raise TypeError
17 ... ast2 = tuple(ast2)
18 ... final_fun(ast2) if final_fun else None
19 ... 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.

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 one containing more than - one containing > 2 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) )
( (((_ out 1) (_ out 2)) ((_ out 3) (_ out 4))) (_ out 5) )
```

Each neighbouring pair or 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.¹⁶ 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 $\mathcal{O}(\log_2(N))$ pyramids.

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 subtraction 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) // 1 - 2 = -1
(1 2 -) // 2 - 1 = +1
```

Finally, the out keyword normally does not allow for output of more than 2 variables. In psll the out keywords 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.

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

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

psll introduces string literals, such that (set s "hello") expands into the above code. 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:

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

such that "hello" = "he" + "llo" = ("h" + "e") + ("l" + "lo") =

Array literals

def keyword

Semi-local

A. Optimisation

B. Sharp edges

Despite authors best efforts, this introduces some sharp edges into the language (see Section III B).

As mentioned at the beginning of Section, the introduction of syntactic sugar into psll introduces some edge cases which one ought to watch out for.

Escape characters Because " is used for strings, and [] for arrays...

IV. EXAMPLE PROGRAMS

A. Pseudorandom number generation

```
1 (set seed 312312)
2 (set div (^ 2 16)) // 16-bit divisor
3 (set prime 7) // Prime divisor
4 // Prime divisor is small because of modulo implementation
5
6 // Generate uniformly distributed random number between 0-1
7 // mod(prime*seed + current, 2^16)
8 (def roll (
9   ... (set seed (+ (* seed prime) 1))
10  ... (loop (<=> (<=> seed div) -1) (set seed (- seed div)))
11  ... (set rand (/ seed div))
12 ))
13
14 // Print 100 such numbers
15 (set i 0)
16 (do (<=> i 100) (
17   ... (roll) (out rand "\n")
18   (set i (+ i 1))
19 ))
```

B. Bubble sort

```
1 (set n (arg 999)) // Make nil value
2
3 // Array to be sorted
4 (set a [3 1 4 1 5 9 2 6 5 3 5])
5
6 // Get array length
7 // This will be: (len a N)
8 (set N 0) // Pointer into the array
9 // Increment pointer until goes off the end
10 (loop (! (= (arg a N) n)) (set N (+ N 1)))
11
12 // Append element of a in position q to b
```

```

13 (def append (set b (+ b (- ((arg a q) n) (n n))))
14 // Usage: (set q ...) (append)
15
16 // Bubble sort the array
17 (do again (
18 ... (set again 0)
19 ... (set p 0) // Position pointer
20 ... (loop (! (! (<=> p (- N 1)))) ( // For all pairs
21 ... (set this (arg a p))
22 ... (set next (arg a (+ p 1)))
23 ... // This and next need swapping
24 ... (set swap (! (<=> (<=> this next) -1)))
25 ... (? swap ( // If swap
26 ... (set again 1) // Will need to go again
27 ... (set b []) // Start b as an empty array
28 ... // Add prefix of a
29 ... (set l 0)
30 ... (loop (= (<=> l p) -1) (
31 ... (set q l) (append)
32 ... (set l (+ l 1))
33 ... ))
34 ... // Add two elements, swapped
35 ... (set q (+ p 1)) (append)
36 ... (set q (+ p 0)) (append)
37 ... // Add suffix of a
38 ... (set l (+ p 2))
39 ... (loop (= (<=> l N) -1) (
40 ... (set q l) (append)
41 ... (set l (+ l 1))
42 ... ))
43 ... (set a b)
44 ... ))
45 ... (set p (+ p 1)) // Increment position pointer
46 ... ))
47 ... (out (* a " " " ") "\n") // Print a
48 ))
49 (out "done")

```

V. CONCLUSIONS AND OUTLOOK

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

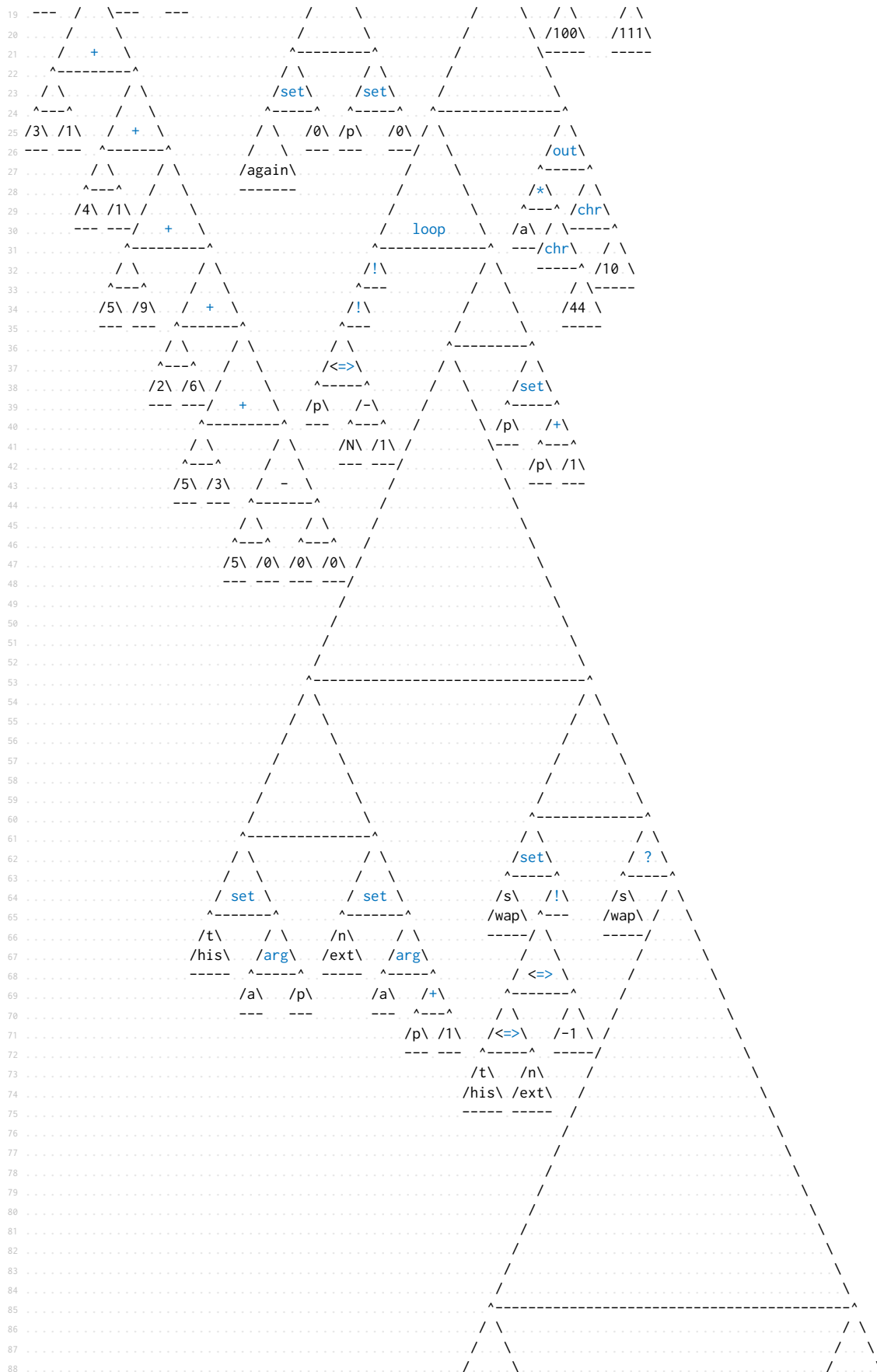
This is by no means a done project, so long as it is a platform for me to learn and have fun. I also believe that the future direction of ps11 poses some genuinely interesting computational problems, namely efficient code optimisation and performing more advanced code transformations. The language is currently not allowing 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 language. This will, however, be a major milestone since the EV structure of the resulting syntax tree will require major 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 ps11 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 mo-

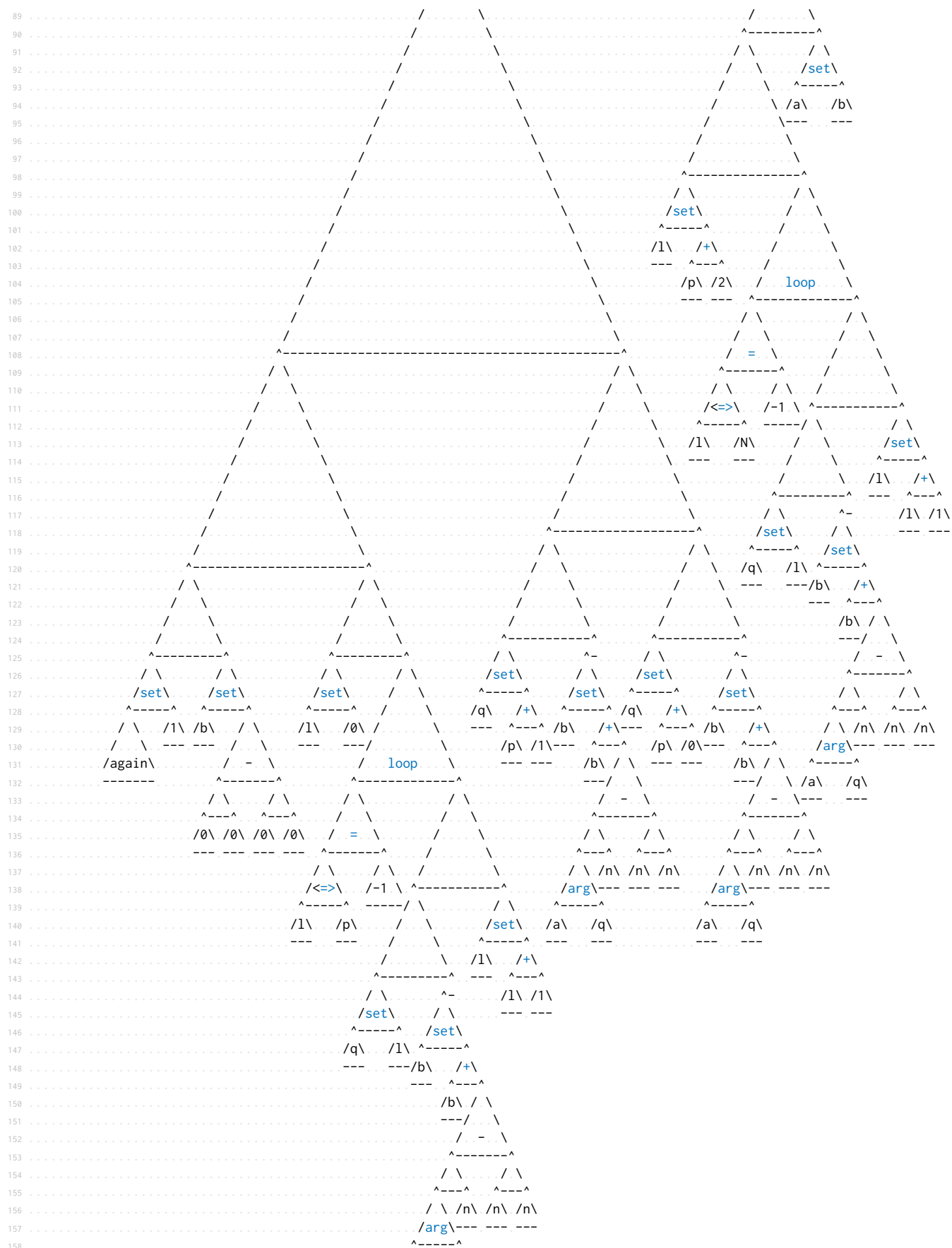
ment. Additional keywords, as well as their supporting architecture will therefore need to be introduced to be able to explicitly specify EV cross-branching structure.

There are a few major parts of ps11 which need to be finished 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 ps11 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 due to be added. These are, for example, len – expanding to the equivalent of line 10 in Listing 4 and nil – initially expanding to (set nil (arg 999)), or something more robust, in the preamble. 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).

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. ps11 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.¹⁷

Finally, I think every programmer shares a certain latent interest in the underlying structure and of the languages they use every day. I would encourage them to scratch that itch. There are plenty of resources to start, but I am inclined to mirror the advice of Casey Muratori:¹⁸ *“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 what you (...) whereas if you read about stuff without ever actually having encountered a problem yet, then you’re just gonna you have no idea [whether its valuable].”* If you really want a starting point though, I recommend David Beazley’s ply and sly projects,^{19–21} which are based on Yet Another Compiler Compiler (YACC).²²





```
159 ..... /a\ ..... /q\  
160 ..... --- ..... ---
```

Listing 4. Bubble sort in Pyramid Scheme. Compiled with `-full-names` and `-co` (considerate optimisation) flags.