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 `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 pharaos, 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.

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.

```
      ^ ^     ^             ^     ^     ^   ^
     ^-/ \   / \           / \   / \   / \  -^
    ^-/out\ /set\         /out\ /out\ /out\ / \
   / \-----^-----^       ^----- ^----- ^----- /out\
  /set\   /x\   /+\     / \   / \   / \   ^-----
 ^-----^  ---  ^---^   /chr\ /chr\ /chr\ /x\
/x\   /#\     /x\ /1\ ^----- ^----- ^-----  ---
---  ^---     --- ---/ \   / \   / \
   /l\           /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.

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@lczyk.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.[12][13] 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`.[14][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 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

Bracket structure In order to assist the programmer in harnessing the power of Pyramid Scheme, we introduce a meta-language - Pyramid Scheme lisp-like notation (`psll`). 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 empty-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` →`(x)` →`(x _ _)`. This also means that keywords with less than two arguments do not need to specify explicit empty-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 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).

```
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)
    if final_fun:
        final_fun(ast2)
    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 (see for an example of such expression). Hence a bracket containing > 2 other brackets gets expanded as follows:

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

is interpreted as:

```
( (((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. 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 height $\mathcal{O}\left(\log_2(N)\right)$.

This macro is applied locally expand to the corresponding pyramid with explicitly specified leaves. Hence the increment is written as `(set x (+ x 1))`. A similar macro allows us to implicitly pad any bracket with underscores from the right, up to the length of 3, such that

The program from Listing 1 can, therefore be written as: can therefore express ethe

usually with writing larger, more powerful Having described the Pyramid Scheme, we now focus on the main subject of this work the

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 (see **[info]** for an example of such expression). Hence a bracket containing > 2 other brackets gets expanded as follows:

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

is interpreted as:

```
( (((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. 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 height $\mathcal{O}\left(\log_2(N)\right)$.

Rolling sum and product

A. Syntactic sugar

The above specification is, in principle, enough to create fully fully functional PS programs. Certain tasks are, however, still rather cumbersome. This section outlines these cases, as well as syntactic sugar constructs introduced to `psll` to alleviate them. All of these are implemented as local (or semi-local) expansion macros, as described in Section **??**. Despite authors best efforts, this introduces some sharp edges into the language (see Section III C).

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:

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

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

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

Array literals

def keyword

Semi-local

B. Optimisation

C. Sharp edges

As mentioned at the beginning of Section III A, 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

```
(set seed 312312)
(set div (^ 2 16)) // 16-bit divisor
(set prime 7) // Prime divisor
// Prime divisor is small because of modulo implementation

// Generate uniformly distributed random number between 0-1
// mod(prime*seed + current, 2^16)
(def roll (
    (set seed (+ (* seed prime) 1))
    (loop (<=> (<=> seed div) -1) (set seed (- seed div)))
    (set rand (/ seed div))
))

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

B. Bubble sort

```
(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 ( // If 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")
```

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 tht the future direction of `psll` 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 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 `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 therefore 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 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 `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 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. `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.[16]

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:[17] *"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,[18–20] which are based on Yet Another Compiler Compiler (YACC).[21]

VERSION NOTES

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

`fd183d296f08e0cba8bf55da907697eaf412f6a7`

and the `psll` repo:[22]

`ea6c6d2e9c8278cc752894a9aff43f7d44bd93a9`

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.[23]

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)

`bash` is the the Dorian Gray of programming languages – timeless.

ACKNOWLEDGEMENTS

I would like to thank Blaine Rodgers and Samuel Hutton for discussions and helpful comments on the manuscript, 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] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

[4] Bryan Cantrill. Zebras All the Way Down. Uptime 2017, https://youtu.be/fE2KDzZaxvE.

[5] Casey Muratori. The Thirty Million Line Problem. https://youtu.be/kZRE7HIO3vk, 2018.

[6] Try It Online! https://tio.run.

[7] Code Golf Stackexchange. https://codegolf.stackexchange.com.

[8] 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`.

[9] Note that the base of the pyramid is a dash (`0x2d`), not an underscore.

[10] Patrick J. Keeling and Jeffrey D. Palmer. Horizontal gene transfer in eukaryotic evolution. *Nature Reviews Genetics*, 2008.

[11] Zachary Weinersmith. Ewok Village of Life. SMBC, https://www.smbc-comics.com/comic/2012-04-08.

[12] 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.

[13] Conor O'Brien. Pyramid Scheme Negation. https://codegolf.stackexchange.com/questions/147513/pyramid-scheme-negation.

[14] 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.

[15] Shriram Krishnamurthi. On the Expressive Power of Programming Languages. PWLConf, https://youtu.be/43XaZEn2aLc, 2019.

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

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

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

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

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

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

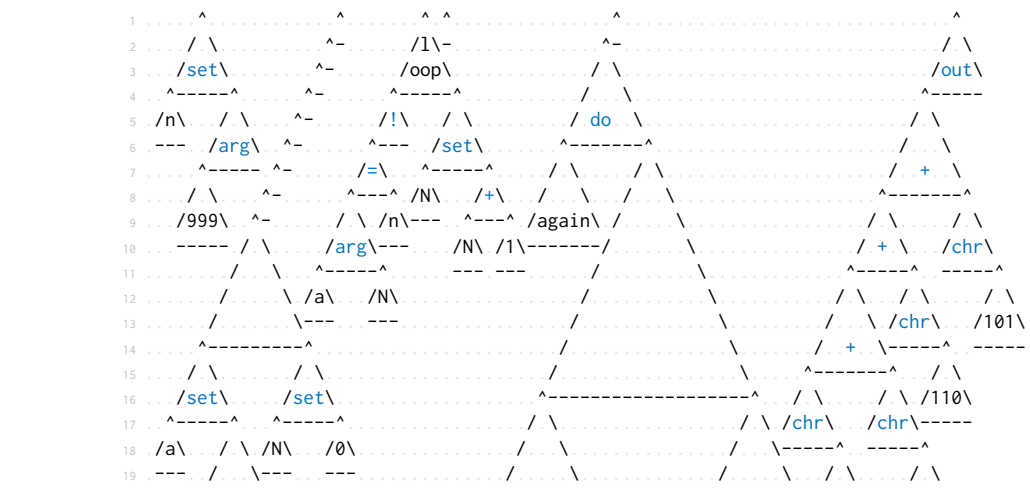
[22] Marcin Konowalczyk. psll-lang. GitHub repository, https://github.com/MarcinKonowalczyk/psll-lang, 2020.

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

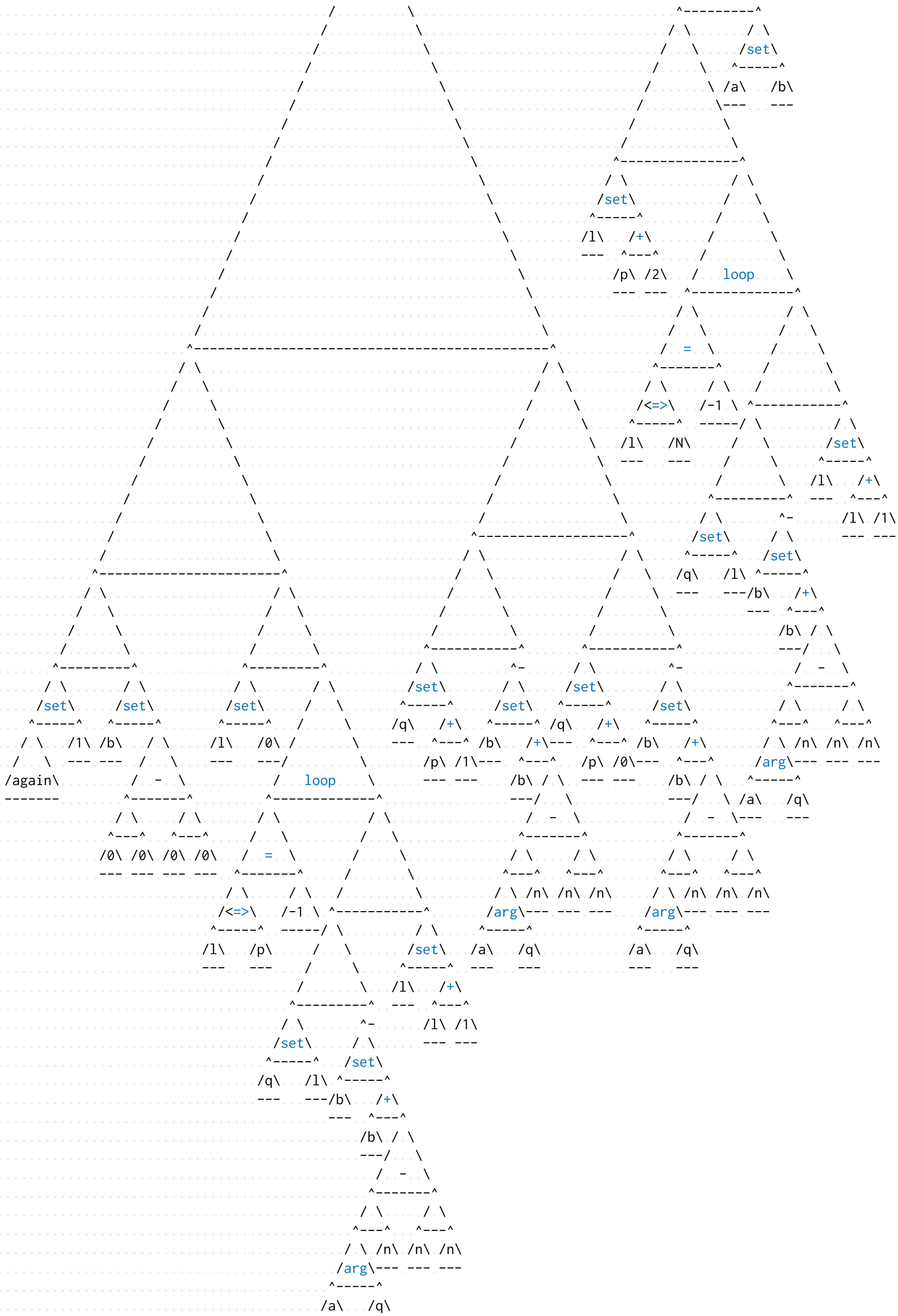
[24] The Lex & Yacc Page. http://dinosaur.compilertools.net.

[25] Blaine Rodgers. High-Octane Rumble Simulation Engine. GitHub repo, https://github.com/PaperclipBadger/high-octane-rumble-simulation-engine, 2017.

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



```
     /     \                     /       \           /       \ /100\   /111\
    /   +   \                   ^---------^         /         \-----   -----
   ^---------^                 / \       / \       /           \
  / \       / \               /set\     /set\     /             \
 ^---^     /   \             ^-----^   ^-----^   ^---------------^
/3\ /1\   /  +  \           / \   /0\ /p\   /0\ / \             / \
--- ---  ^-------^         /   \  --- ---   ---/   \           /out\
        / \     / \       /again\             /     \         ^-----^
       ^---^   /   \      -------            /       \       /*\   / \
      /4\ /1\ /     \                       /         \     ^---^ /chr\
      --- ---/   +   \                     /   loop    \   /a\ / \-----^
            ^---------^                   ^-------------^  ---/chr\   / \
           / \       / \                 /!\           / \    -----^ /10 \
          ^---^     /   \               ^---          /   \       / \-----
         /5\ /9\   /  +  \             /!\           /     \     /44 \
         --- ---  ^-------^           ^---          /       \    -----
                 / \     / \         / \           ^---------^
                ^---^   /   \       /<=>\         / \       / \
               /2\ /6\ /     \     ^-----^       /   \     /set\
               --- ---/   +   \   /p\   /-\     /     \   ^-----^
                     ^---------^  ---  ^---^   /       \ /p\   /+\
                    / \       / \     /N\ /1\ /         \---  ^---^
                   ^---^     /   \    --- ---/           \   /p\ /1\
                  /5\ /3\   /  -  \         /             \  --- ---
                  --- ---  ^-------^       /               \
                          / \     / \     /                 \
                         ^---^   ^---^   /                   \
                        /5\ /0\ /0\ /0\ /                     \
                        --- --- --- ---/                       \
                                      /                         \
                                     /                           \
                                    /                             \
                                   /                               \
                                  ^---------------------------------^
                                 / \                               / \
                                /   \                             /   \
                               /     \                           /     \
                              /       \                         /       \
                             /         \                       /         \
                            /           \                     /           \
                           /             \                   ^-------------^
                          ^---------------^                 / \           / \
                         / \             / \               /set\         / ? \
                        /   \           /   \             ^-----^       ^-----^
                       / set \         / set \           /s\   /!\     /s\   / \
                      ^-------^       ^-------^         /wap\ ^---    /wap\ /   \
                     /t\     / \     /n\     / \        -----/ \      -----/     \
                    /his\   /arg\   /ext\   /arg\           /   \         /       \
                    -----  ^-----^  -----  ^-----^         / <=> \       /         \
                          /a\   /p\       /a\   /+\       ^-------^     /           \
                          ---   ---       ---  ^---^     / \     / \   /             \
                                              /p\ /1\   /<=>\   /-1 \ /               \
                                              --- ---  ^-----^  -----/                 \
                                                      /t\   /n\     /                   \
                                                     /his\ /ext\   /                     \
                                                     ----- -----  /                       \
                                                                 /                         \
                                                                /                           \
                                                               /                             \
                                                              /                               \
                                                             /                                 \
                                                            /                                   \
                                                           /                                     \
                                                          /                                       \
                                                         /                                         \
                                                        ^-------------------------------------------^
                                                       / \                                         / \
                                                      /   \                                       /   \
                                                     /     \                                     /     \
                                                    /       \                                   /       \
```



set
/a\ /b\
set
/l\ /+\
/p\ /2\
loop
=
/<=>\ /-1
/l\ /N\
set
/l\ /+\
/l\ /1\
set
set
/q\ /l\
/b\ /+\
/b\
-
set set set set set set
/q\ /+\ /q\ /+\
/n\ /n\ /n\
/l\ /b\ /l\ /0\ /b\ /p\ /1\ /b\ /p\ /0\ /b\
arg
/again\
- loop - -
/a\ /q\
/0\ /0\ /0\ /0\
=
/n\ /n\ /n\ /n\ /n\ /n\
/<=>\ /-1
arg arg
set
/a\ /q\ /a\ /q\
/l\ /p\
/l\ /+\
/l\ /1\
set
set
/q\ /l\
/b\ /+\
/b\
-
/n\ /n\ /n\
arg
/a\ /q\

```
                                                                                                                    ---   ---
```

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