University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Small-step Boogie Woogie: An Operational Semantics for the Piet Language

Mateusz Alchimowicz

March 27, 2025

Abstract

Piet is a visual esoteric programming language that uses the difference between colours to calculate operations. This requires unique ways to specify language execution. The online specification is vague and imprecise with no formal definitions. This project comprehensively formalises the language to create an unambiguous specification. Small-step semantics operational semantics were used to write the formalisation and OCaml was used to create a definitional reference interpreter. It was found that the reference interpreter performed better than most pre-existing interpreters.

Acknowledgements

Thank you to my supervisor, Dr Simon Fowler, for his invaluable support and guidance throughout.

I would like to also thank my friends and family for their support during this project.

Education Use Consent

Contents

1 Introduction

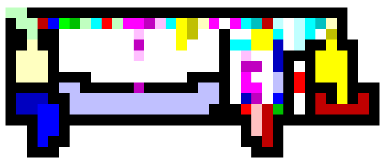

Figure 1.1: Piet code that calculates one number to the power of another, by François Tavin (Morgan-Mar 2023a).

1.1 Motivation

Piet is a visual esoteric programming language created by David Morgan-Mar. It relies on colour and movement to execute code, with no characters used at all. He provides a specification on his website (Morgan-Mar 2018). While being comprehensive, and having been updated to clear up confusion, the specification remains purely in written English. This allows for ambiguity and misunderstandings to arise due to the nature of language. For example, Roll is one of the 18 operations the language has. Morgan-Mar's specification describes it as follows:

> "roll: Pops the top two values off the stack and "rolls" the remaining stack entries to a depth equal to the second value popped, by a number of rolls equal to the first value popped. A single roll to depth n is defined as burying the top value on the stack n deep and bringing all values above it up by 1 place. A negative number of rolls rolls in the opposite direction. A negative depth is an error and the command is ignored. If a roll is greater than an implementation-dependent maximum stack depth, it is handled as an implementation-dependent error, though simply ignoring the command is recommended." (Morgan-Mar 2018)

As a specification for an operation, this language is vague, imprecise and convoluted. Implementations of Piet do exist, but it is unclear how one implementation compares to another. It also unclear whether these interpreters conform with the specification. Therefore, there exists a need to provide an unambiguous specification of Piet to solve these issues. Programming languages techniques are designed for text-based languages, so this project also aims to see how successful these techniques are when dealing with the unique paradigm Piet presents. To this end, a reference interpreter was created following the semantics to allow for evaluation.

This paper demonstrates that existing techniques can be mapped onto unique languages like Piet, with operational semantics being flexible enough to encompass Piet's complex rules surrounding program execution. This provides confidence in operational semantics being a useful technique in creating and describing many types of programming languages. This is proven through the evaluation of the reference interpreter which, in implementing the semantics, provides equal to better results than other pre-existing Piet interpreters.

1.2 Summary

The paper is laid out in the following way:

- **Chapter 2** - An overview of what esoteric languages are, what Piet is, and what operational semantics are, with examples and provides an explanation of what this project aims to achieve.
- **Chapter 3** - The operational semantics for Piet. This includes rules, necessary functions and formal definitions.
- **Chapter 4** - The implementation of a reference interpreter that follows the operational semantics.
- **Chapter 5** - The evaluation of the interpreter against pre-existing Piet interpreters, and a discussion of the success of the project.
- **Chapter 6** - The conclusion, possible future work, and a personal reflection on the project.

2 | Background

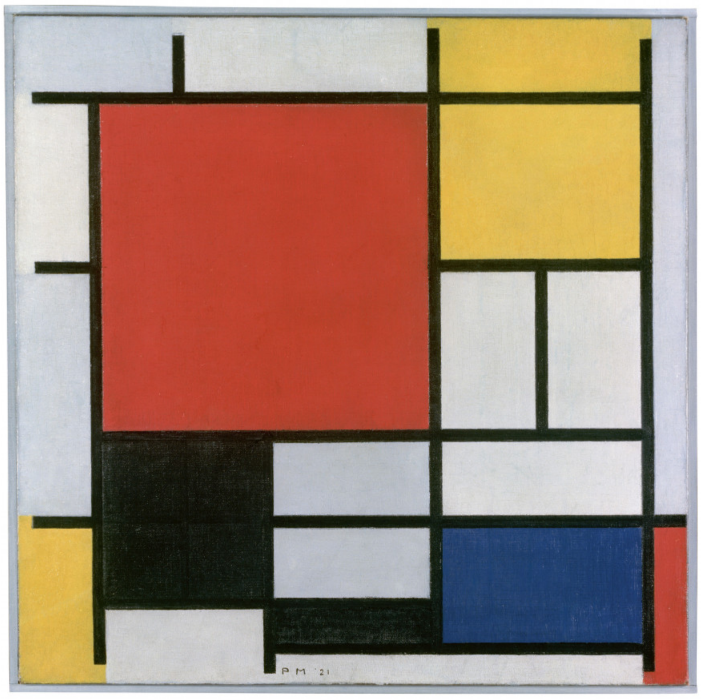

Figure 2.1: *Composition with Large Red Plane, Yellow, Black, Grey and Blue, by Piet Mondrian (Mondrian 1921).*

2.1 Piet Mondrian

Piet Mondrian (1872 - 1944) was a Dutch artist and painter. Mondrian was part of the De Stijl group of Dutch artists, who were influenced by the contemporary cubist and Bauhaus movements (Lewis 1957). Mondrian described his style as neoplasticism. This style is what is associated with his most famous pieces of work today. Turner describes it as:

> “A grid, delineated by black lines, was filled with blocks of primary colour [...] The evanescence of natural shapes was reduced to a few essential expressive means: horizontal and vertical lines, areas of primary colour and black and white.” (Turner 1996, p. 749)

His neoplastic work is stark and bold. It abstracts away most of human presence within what he depicts, as in Figure 2.1. But it still leaves the visible, man-made cityscape behind. Lewis describes Mondrian’s work as leaving “separate — yet interdependent” (Lewis 1957) units.

2.2 Esoteric Languages

There exists a subset of programming languages called “Esoteric programming languages”, “esolangs” or alternatively, “Weird programming languages”. These languages push what may be possible to achieve in the field of programming languages. Most are not feasible to code in seriously, with some making the task almost impossible to undertake. Michael Mateas categorises esoteric languages as follows:

“[Esoteric or Weird] languages are considered in terms of four dimensions of analysis: (1) parody, spoof, or explicit commentary on language features, (2) a tendency to reduce the number of operations and strive toward computational minimalism, (3) the use of structured play to explicitly encourage and support double-coding and (4) the goal of creating a puzzle, and of making programming difficult.” (Mateas 2008, p. 267)

2.2.1 INTERCAL

INTERCAL is widely-held as the first esoteric programming language, created in 1972 by Don Woods and James Lyon (Mateas 2008, p. 267). Following Mateas’ categories, INTERCAL is a parody esoteric language with minimalist and puzzle aspects. INTERCAL, an acronym of “Compiler Language With No Pronounceable Acronym”, aims to be a language which “ha[d] nothing at all in common with any other major language.”(Lyon and Woods 1973, p. 2) To this end, INTERCAL implements 5 operators, 2 binary (mingle “¢” and select “~”) and 3 unary (AND “&”, OR “V” and XOR “Ṽ").

INTERCAL also implements the “PLEASE” qualifier, which the programmer must use the correct amount of times. Too few PLEASEs in a given program and it will not compile due to the programmer’s impoliteness. Too many PLEASEs will also cause the program to not compile as INTERCAL perceives the programmer as too obsequious. Provided in Tonsil A of the Reference Manual is a list of all characters used in INTERCAL (Lyon and Woods 1973, p. 20). As is evident in Tonsil A, INTERCAL parodies the APL language by using hard to produce characters to represent operators.

```
DO :1 <- #0$#256
```

Figure 2.2: Code in Intercal to store 65536 in a 32-bit variable.

2.2.2 Malbolge

While INTERCAL makes coding difficult, languages like Malbolge make coding impossible. Malbolge is a minimalist, puzzle esoteric programming language. Named after Dante’s eighth circle of Hell (Olmstead 1998), Malbolge’s express purpose is to be a difficult programming language. Malbolge, unlike most programming languages, works in trinary. The trinary CPU has three registers, A (Accumulator), C (Code Pointer), D (Data Pointer), all are automatically incremented after each instruction. Given an ASCII character, Malbolge uses this formula,

$$\texttt{char} - 33 + \texttt{C} \pmod{94}$$

to cross-reference the character in a table of characters, potentially matching it to a character representing an operator. After the instruction, 33 is subtracted from the value in C, which is then cross-referenced against another table of characters. The character’s value is placed in C, which is then incremented. Given the infernal complexity of the language, it took two years for an algorithm to create a "Hello, World!" program (Mateas 2008, p. 272). Even now, Malbolge remains a language hard to code in and hard to explain.

```
(=<‘$9]7<5YXz7wT.3,+O / oK%$H~D|#z@b=‘{^Lx8%$Xmr
    kpohm-kNi;gsedcba‘_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>=<;:9876543s+O<oLm
```

Figure 2.3: "Hello, World!" in Malbolge (Mateas 2008, p. 267).

2.3 Piet

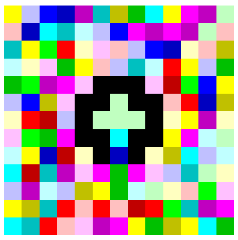

Figure 2.4: GIF image that prints "Hello, World!", by Thomas Schoch (Morgan-Mar 2023a).

Piet, the focus of this paper, is an esoteric programming language, created by David Morgan-Mar. Morgan-Mar has created other esoteric language, including Chef (Morgan-Mar 2022). Both Chef and Piet fall into the category of double-coding esoteric languages. Double-coding refers to program code that approaches the format of another type of document. Chef, for example, requires all programs to be in the form of a recipe.

Piet takes inspiration from Mondrian's work and as such is a two-dimensional visual language, relying on colours and shapes to parse code. Piet code takes the form of a rectangular image, with execution starting from the top-left corner. Colours exist on two cycles: hue, and shade. The difference between one colour's hue and shade and another's provides the difference, or delta (Δ). Most colours have a hue and shade component.

Table 2.1: The 16 vibrant colours available in Piet. The colours sit on the two cycles. For example, Light Red to Dark Magenta is a a hue change of 5 and a shade change of 2, i.e a Δ of (5,2). The cycles only operate in one direction, so Dark Magenta to Light Red is a Δ of (1,1)

Hue / Shade	0	1	2	3	4	5
0	Light Red	Light Yellow	Light Green	Light Cyan	Light Blue	Light Magenta
1	Regular Red	Regular Yellow	Regular Green	Regular Cyan	Regular Blue	Regular Magenta
2	Dark Red	Dark Yellow	Dark Green	Dark Cyan	Dark Blue	Dark Magenta

Two additional colours exist outside of the hue and shade cycles. These two create a subset, which this paper calls tones: black and white. Colours are the units from which programs are built. A

Table 2.2: The 18 operations available in Piet categorised by their Δ. Hue difference is along the rows, shade difference is along the columns.

Δ shade \ Δ hue	0	1	2	3	4	5
0	`no_op`	`add`	`divide`	`greater`	`duplicate`	`in_char`
1	`push`	`subtract`	`modulo`	`rotate`	`roll`	`out_int`
2	`pop`	`multiply`	`not`	`switch`	`in_int`	`out_char`

single unit of a colour is called a codel. Codels can form codel blocks, or regions, with codels of the same colour. Regions follow 4-directional contiguity. This means two codels fall into the same region if a line moving only up, down, left or right can connect them without changing colours.

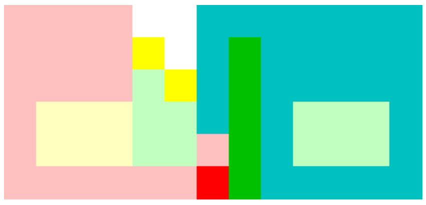

Figure 2.5: Piet code that has several colour regions. There is a light green region completely surrounded by a dark cyan region. This dark cyan region has a limb which extends into the middle. As cyan codels all touch, this region is contiguous. The two yellow codels are diagonal from one another. They do not touch and so do not form a region together.

Execution of Piet code starts in the top-left hand corner for every image. Execution is controlled by the direction pointer (dir) and the codel choice (cc), which together this paper calls the pointer. The direction pointer can point up, down, left or right. When executing code, the pointer tries to find a codel at the edge of the current region furthest in the direction of the direction pointer (i.e the rightmost codel for a direction pointer pointing right). It moves itself to that edge and then attempts to step across the edge into the next colour region. The difference in the previous region's colour and the new region's colour is the delta, this provides the operation to calculate. The codel choice represents an either-or preference when two or more codels are the furthest from the pointer. Figure 2.6 illustrates an example.

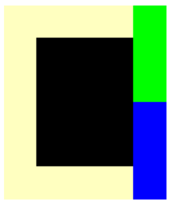

Figure 2.6: With the direction pointer facing right, the execution could either move up into the Green region or down into the Blue region as the edge points are equally far away. The codel choice represents which decision should be made. Since the pointer rotates, it may be easier to visualise this as the relative left or right based on the pointer's direction.

The tones are unique from the other colours as they exhibit special properties. White has the pointer step codel by codel instead of crossing the region in one go. White also has a Δ of (0,0) with any colour. Black is not traversable. The pointer cannot step onto Black codels and if code starts with Black in the top-left hand corner it is untraversable. It is therefore impossible to calculate the Δ of Black with any other colour.

Execution cannot continue beyond the edge of an image, nor can it enter a Black region. When this occurs, the codel choice should be flipped. If this does not allow execution to continue, the direction pointer should rotate clockwise once. This continues until execution resumes or 8 attempts at resuming execution fail. After 8 attempts, execution halts.

Piet exhibits double-coding due to its visual nature. Every program is itself an image, in either .png or .gif format. Aesthetic value of each Piet code-image is subjective, but the freeform nature of the language allows for structured play to create visually stunning, and functional code. Piet has no variables, instead, it uses a stack to store integers. The 18 operations Piet implements modify the stack in some way. This is all the programmer has at their disposal. These 18 operations are rudimentary, and as such allow for multiple different ways of reaching the same result.

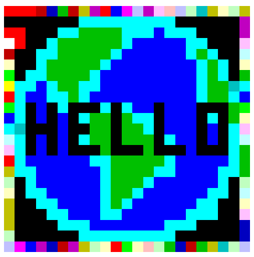

Figure 2.7: *Hello Globe that prints "Hello, World!", by Kelly Boothby (Morgan-Mar 2023a).*

Figure 2.4 and Figure 2.7 are both implementations of "Hello, World!". Both reach the same result, and yet look remarkably different.

2.3.1 Example execution

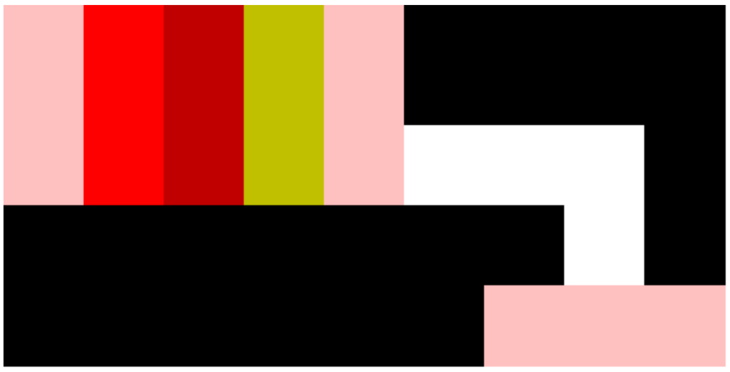

Figure 2.8: *Piet code which adds 2 and 2 together and outputs the result, by Jens Bouman (Bouman 2024).*

Figure 2.8 is a program which calculates $2 + 2$. Execution is as follows:

1. Piet code starts in the top-left corner. Initially, the direction pointer points right, with codel choice set to 0. This represents left relative to the direction of the pointer. The codel it is currently on touches the codel below it, and both are of the same colour. This means they are in the same region.
2. The pointer attempts to move. Since the codel the pointer it is on and the one below it are equally as far to the right, the codel choice is used to break the tie. The codel the pointer

is currently on is the most left, relative to the pointer's direction. So, the pointer simply moves one codel forward onto the regular red.

3. The Δ between Light Red and Regular Red is $(0, 1)$. This means push the size of the previous region (2) onto the stack. Movement is attempted with the same result as before, the pointer steps forward one.
4. The Δ between Regular Red and Dark Red is $(0, 1)$, so the region size (2) is again pushed onto the stack. Movement occurs as before.
5. The Δ between Dark Red and Dark Yellow is $(1, 0)$ which is add. The top two values are popped off the stack, then added together and the result (4) is pushed back on. Movement, again, occurs as before.
6. The Δ between Dark Yellow and Light Red is $(5, 1)$, which is output number. The top value is popped off and printed.
7. Then, movement is attempted, but blocked by the Black codel, so the codel choice is toggled. This means that the rightmost codel, relative to the direction pointer should be chosen. This leads the pointer to move onto the White.
8. The Δ between White and any colour is $(0, 0)$, so no operation is executed. When on White, the pointer moves codel by codel, until it hits something. In this case, a Black codel. The codel choice and direction pointer are toggled, meaning the pointer is facing down.
9. Execution continues codel by codel, until the pointer moves onto the Light Red. The pointer then moves to the attempts to move out of this block, by finding the edge codels, but cannot. The pointer continues to toggle the direction pointer and codel choice, but execution does not resume.
10. These failed movement attempts happen 8 times before execution halts.

2.4 Operational Semantics

Designing programming languages requires creating a rigorous specification which formalises the language. This often includes a grammar, and example usage to define the types, control flow and structure of the language. Strategies vary, however. For example, The GNU C Reference Manual provides a short definition of character meanings, before providing written explanations, alongside example code to describe its implementation of C (*The GNU C Reference Manual* n.d.).

An alternative way to describe the control flow and operation of a programming language is to formalise its semantics. WebAssembly presents its specification using Extended Backus-Naur Form (EBNF) grammars and sets of operational semantic rules (Haas et al. 2017). Plotkin, in *A structural approach to operational semantics*, describes operational semantics as a way of describing languages as set of configurations, with the states being a term, or a set of terms. Languages move between these configurations by using a transition relation, which, through simplification of a given term, provides either a new state, or halts the program (Plotkin 2004, p. 4). This is specifically small-step operational semantics, where terms are simplified. An alternative exists, big-step semantics, where terms are evaluated immediately to a value, a minimal unit which cannot be evaluated further. Figure 2.9 is an example taken from *Types and Programming Languages*.

The arrow ($\rightarrow$) represents the transition relation, taking one step to simplify the term, but crucially not evaluating it to a value. The box in the top right represents the judgement, which states that in one step, the program can change its state from `t` to a new term, `t`'. Figure 2.9b presents 2 axioms and one rule. The first axiom states that if the predicate of the conditional is true, the state should move to `t`$_2$. If the predicate is false. the state should instead move to term t_3. These are axioms, as they have no premises, statements above the line that must hold true for the statement under the line to do so. In other words, these always hold true. The final rule states that if `t`$_1$ can be simplified to `t'`$_1$, the conditional can be simplified in the same way. This does not evaluate the

`t` ::=		*terms:*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v` ::=		*values:*
	`true`	*true value*
	`false`	*false value*

(a) Syntax of boolean expressions

$$\boxed{\texttt{t} \longrightarrow \texttt{t'}}$$

$$\frac{}{\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_2}$$

$$\frac{}{\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{t}_3}$$

$$\text{IF}\;\frac{\texttt{t}_1 \longrightarrow \texttt{t'}_1}{\texttt{if } \texttt{t}_1\texttt{then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \longrightarrow \texttt{if } \texttt{t'}_1\texttt{then } \texttt{t}_2 \texttt{ else } \texttt{t}_3}$$

(b) Operational Semantic rules for boolean expressions

Figure 2.9: *Operational Semantics of boolean expressions. (Pierce 2002, p. 34)*

result of the if conditional, but explains how programs taking a step in simplifying the predicate can do the same for the conditional.

2.4.1 WebAssembly

WebAssembly is a modern instruction format to allow programming languages to be compiled for use on the web (Haas et al. 2017). WebAssembly is currently being developed by the World Wide Web Consortium (W3C) in concert with the owners of the most used web browsers. Since the language's goal was to be used as a standard across multiple browsers, the language required a system of formalising the workings of the language. This would allow versions of WebAssembly to be written that work within a specific browser's ecosystem, whilst allowing interoperability of code between browsers. To achieve this, WebAssembly's formal specification is written using small-step operational semantics. Figure 2.10 explains how unary operations

$$\frac{}{C \vdash t.unop : [t] \rightarrow [t]}$$

(a) WebAssembly's unary operation's typing rule.

$$\begin{array}{llll} (t.\mathbf{const}\ c_1)\ t.unop & \hookrightarrow & (t.\mathbf{const}\ c) & (\text{if } c \in unop_t(c_1)) \\ (t.\mathbf{const}\ c_1)\ t.unop & \hookrightarrow & \mathbf{trap} & (\text{if } unop_t(c_1) = \{\}) \end{array}$$

(b) Reduction rules on the application of unary operations in WebAssembly.

Figure 2.10: *Typing rule and reductions for unary operations in WebAssembly (*WebAssembly Specification *2025, p. 33, p. 88).*

work in WebAssembly. Figure 2.10a is an axiom, describing that, for a unary operation to be valid, it must take a parameter of a type t, and return a value of the same type. As it is an axiom, there are no premises that need to hold for it to be true, i.e it is always true. Figure 2.10b presents a reduction, explaining how unary operators should be implemented. It is not necessary to affirm c_1's type as the axiom ensures this. Line 1 states to pop the value off of the stack, and if the c_1 is within the operator's domain, wrap the resulting value c and push it back onto the stack. Line two states if the operator cannot be applied to c_1, the system should return a trap, an instruction to halt execution. WebAssembly was awarded the 2021 ACM SIGPLAN Programming Languages Software Award, stating:

> "WebAssembly's formalization effort both shows the benefits of bringing PL techniques to widely-adopted languages—the disparate browser implementations all worked correctly with no deviation from the specification—and makes WebAssembly fertile ground for future research on web languages." - (*Programming Languages Software Award* n.d.)

Semantics are not exclusively limited to normal languages. Cameron Wong presented a formal semantics for Befunge, a 2-dimensional esoteric language at SIGBOVIK 2019 (Wong 2019). His paper presents that semantics can be applied to Befunge, including to the 2-dimensional movement system the language uses.

2.5 Discussion

As mentioned in Section 2.2, esoteric programming languages occupy a niche in the programming space. This can mean that unique and innovative ways of facilitating coding can be easily written off as a joke or parody. While a visual language like Piet is unlikely to be any programmer's first choice, this means that formal techniques are not often used to describe the functioning of any of these languages, leaving much up to interpretation. However, this seems to be part of the fun and challenge of esoteric language. Morgan-Mar seems to lean into this too, admitting "whatever makes the most sense" (Morgan-Mar 2018) is often the right answer to any question a would-be Pietist would have. This leaves space to experiment with "formalising" the specification, with an aim to give more intelligible, unambiguous definitions of the language's operations and control flow structure.

Formalising Piet through programming language techniques leaves many possibilities still open. However, reflecting on WebAssembly's success in using operational semantics to formalise a language and achieve consistency between different interpretations, it seems prudent to do the same with Piet. A more formal specification creates the possibility of implementations of Piet that preserve the same high-level functioning, whilst leaving implementation-specific details to interested programmers who would like to take on the task. This strategy tries to preserve, therefore, *the whatever makes the most sense* approach to parts of Piet whilst providing an unambiguous baseline for interpreters and IDEs to use as a foundation for their programs.

As mentioned in Section 2.4, there exists two types of operational semantics, big-step and small-step semantics. It is hard, conceptually, to quantify how Piet may evaluate code to a value, given the non-textual nature of the language and its reliance on 2-dimensional movement, as opposed to the more linear movement seen in more normal programming languages. Wong's earlier work on Befunge, however, provides a foundation from which Piet's formalisation can be explored. As the language relies almost purely on its state, explaining how Piet transitions between its possible states based on the current state would be a better solution. Consequently, this formalisation will use small-step semantics.

Piet interpreters already exist, with a selection listed on the language's website (Morgan-Mar 2023*b*). As a result, in order to evaluate the formalisation's success, a reference interpreter should be created and tested against extant interpreters. It isn't always possible to assume that a given

Piet image runs as expected. This means success should be evaluated against how similar the reference interpreter's output for a given image compares to the evaluation set. For example, if code fails or outputs an unexpected result in the evaluation set, correct execution would mean outputting the same incorrect result.

It may be possible, as well, to consider a type system for Piet, as the colours and tones each interact differently when executing movement. Another additional idea was an extension to Piet, potentially including both new colours and operations, which would require more semantic rules. While these two ideas have much possibility, the scope was kept narrow to fit into the 20-credit requirements of the project.

Therefore, the aim of the project is to explore the potential to formalise the Piet language using programming language techniques, including small-step operational semantics. A reference interpreter should also be created for evaluation purposes.

3 Operational Semantics

3.1 Approach

Operational semantics usually relies on providing rules to simplify or evaluate terms. These rules form part of the formalisation, including a grammar, a parser and lexer to tokenise the characters into expressions.

When a language does not use statements or expressions per se, it can be hard to map these concepts onto the language. For example, semantics rules can explain how a subroutine should be run. The rule abstracts away the need to consider how the interior machinery of the program moves to run the subroutine. This is too high-level for Piet. The idea of a subroutine itself is too abstract. The rudimentary equivalent of a subroutine would appear as a self contained segment of the image that the pointer must move towards. Iteration, too, must be carefully laid out as a segment of codels connected together in a loop. In this way, Piet is similar to Assembly and byte-code. Piet, however, makes no use of external stores and has no mechanism to allow for jumps.

Fundamentally, this is why the rules for describing Piet are movement based. Correct movement leads to correct execution. So, providing the outline of what correct movement looks like, and what causes execution to halt allows for other programmers to create their own internal functions whilst preserving correct movement.

3.2 Semantics

This section will present the formalisation of Piet in a sequential manner. It will start with an Extended Backus-Naur Form grammar, before enumerating mappings, the definitions of operations, auxiliary functions and then the small-step semantic rules. It also covers considerations and decisions made when attempting to use programming language techniques on a language that falls outside of the ordinary.

To allow ease of reference, tables used earlier are repeated in this section.

3.3 Grammar

At any given codel on an image, execution could vary wildly based on what direction the direction pointer points and which way the codel choice is set. This is a unique challenge that visual languages may exhibit. As a consequence, it may seem useful to preserve as much information as possible about Piet's state. This however, increases overhead, and may include state variables that in actuality rely on another value stored in the state tuple. This would lead to obfuscation of the key values which guide execution.

As a result, a minimalist approach was taken, paring down potential state variables to a minimal tuple. This approach aims to keep the essentials clear and provide the minimum necessary to understand program flow. This may increase the need for auxiliary functions, but these would be called on an as-when basis, rather than being part of the state. Figure 3.1 details the grammar.

n	::=	$n \in \mathbb{Z}$	*integers*
h	::=	Red | Yellow | Green | Blue | Cyan | Magenta	*hues*
s	::=	Light | Regular | Dark	*shades*
t	::=	White | Black	*tones*
k	::=	(h,s) | t	*colours*
op	::=	no_op | push | pop	*operations*
		| add | subtract | multiply	
		| divide | modulo | not	
		| greater | rotate | switch	
		| duplicate | roll | in_int	
		| in_char | out_int | out_char	
dir	::=	right | down | left | up	*Direction Pointer*
cc	::=	0 | 1	*Codel Choice*
ptr	::=	(dir,cc)	*Pointer*
pos	::=	(n, n)	*Position*
inc	::=	n	*Increment Counter*
$\overrightarrow{V}$	::=	$\epsilon \mid V \cdot \overrightarrow{V}$	*Stack*
size	::=	n	*Codel Size*
img	::=	$[n \times n]$	*Image*
$\mathcal{S}$	::=	{img,size,$\overrightarrow{V}$,ptr,pos,inc}	*State*

Figure 3.1: *An EBNF Grammar for Piet*

All code images are made up of colours only. Piet does store integers on the stack, however, and so they are a value the language uses to execute operations. While the language can receive a character input and produce a character output, internally characters are stored as integers. A colour value is either made up of a hue and a shade, or it is a tone. `op` details the 18 operations available in Piet.

`img` is a 2 dimensional array of colour values, representing every possible pixel in an image. The array traverses the rows of the image before the columns, with (0,0) as the top left corner. The value of a position is larger the further right or down it moves. The codel size is an integer value that states the length of a codel in pixels. A codel size of 1 means a codel is equivalent to a pixel. A codel size of 2 means a codel is a 2x2 square of pixels. The image array and codel size do not change during execution.

The direction pointer and codel choice together form the pointer tuple. Position represents the current location of the pointer as a 2-tuple of a (row,col) position. The increment counter records how many failed movement attempts there have been since the last successful movement. This is used when halting execution, and incrementing the pointer. The stack is a store of integers following the usual definition of the stack abstract data type. It can either be empty, ϵ, or an ordered collection of integer values, $\overrightarrow{V}$. The interpunct ($\cdot$) represents the cons operation.

Finally, the state tuple collates all the values necessary to represent a given state for any Piet program at any given time.

3.4 Mappings

Since Piet uses no words, this formalisation eschews parsers, lexers, or syntax in the normal sense. There is, however, the need to convert an image with RGB values to an array of Colour values.

There are, therefore, several mappings used in Piet, including from RGB colour space to Piet's internal colour representation, and from a given Δ value to an operation. These are listed below.

Table 3.1: *The 18 colours of Piet, including RGB colour values.*

Shade \ Hue	0	1	2	3	4	5
0	Light Red #FFC0C0	Light Yellow #FFFFC0	Light Green #C0FFC0	Light Cyan #C0FFFF	Light Blue #C0C0FF	Light Magenta #FFC0FF
1	Regular Red #FF0000	Regular Yellow #FFFF00	Regular Green #00FF00	Regular Cyan #00FFFF	Regular Blue #0000FF	Regular Magenta #FF00FF
2	Dark Red #C00000	Dark Yellow #C0C000	Dark Green #00C000	Dark Cyan #00C0C0	Dark Blue #0000C0	Dark Magenta #C000C0

	#FFFFFF White		#000000 Black

There are 16 hue-shade colours in Piet. This table provides a Piet representation for any RGB Code. The Piet specification allows handling of additional colours to be interpreter-dependent. This formalisation instead takes the approach that additional colours are internally represented as Black.

There are the 18 operations available in Piet, each operating on the stack as described in the following section. Table 3.2 mapping turns a Δ value into an operation.

Table 3.2: *The 18 operations available in Piet categorised by their delta value. Hue difference is along the rows, shade difference is along the columns.*

Δ shade \ Δ hue	0	1	2	3	4	5
0	`no_op`	`add`	`divide`	`greater`	`duplicate`	`in_char`
1	`push`	`subtract`	`modulo`	`rotate`	`roll`	`out_int`
2	`pop`	`multiply`	`not`	`switch`	`in_int`	`out_char`

3.5 Operations

The operations are currently explained in English text only. This increases cognitive overhead and grappling with ambiguous definitions can lead to differing implementations. Therefore, operations are expressed in this section in a method approaching mathematical definitions. This provides each operation with an unambiguous definition that presents the information without any additional overhead.

3.5.1 calc Function

calc`(op,pos,`$\vec{V}$`,ptr)` $\rightarrow$ `(`$\vec{V'}$`,ptr')`

The **calc** auxiliary function takes an operation, position, current stack, and pointer. It calculates the result and returns the updated pointer and stack. This allows for semantic definitions to be created for all operations, detailing how they should work, given a pointer, position and stack. The definitions for the operations are provided in Section 3.5.2, with an explanation following after.

3.5.2 Definitions

$$
\begin{aligned}
&\text{calc}(\texttt{op},\texttt{pos},\vec{V},\texttt{ptr}) &&= (\vec{V},\texttt{ptr})\\
&\textit{Where:} && |\vec{V}| = 0 \land \texttt{op} \notin \{\texttt{push},\texttt{in\_char},\texttt{in\_int}\}\\
&\text{calc}(\texttt{op},\texttt{pos},\vec{V},\texttt{ptr}) &&= (\vec{V},\texttt{ptr})\\
&\textit{Where:} && |\vec{V}| < 2 \land \texttt{op} \in \{\texttt{add},\texttt{subtract},\texttt{divide},\\
& && \texttt{multiply},\texttt{modulo},\texttt{greater}\}\\
&\text{calc}(\texttt{no\_op},\texttt{pos},\vec{V},\texttt{ptr}) &&= (\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{push},\texttt{pos},\vec{V},\texttt{ptr}) &&= (\texttt{n}\cdot\vec{V},\texttt{ptr})\\
&\textit{Where: } n &&= |\texttt{search(pos)}|\\
&\text{calc}(\texttt{pop},\texttt{pos},n\cdot\vec{V},\texttt{ptr}) &&= (\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{switch},\texttt{pos},n\cdot\vec{V},(\texttt{dir},\texttt{cc})) &&= (\vec{V},(\texttt{dir},(\texttt{cc+n})\%2))\\
&\text{calc}(\texttt{rotate},\texttt{pos},n\cdot\vec{V},(\texttt{dir},\texttt{cc})) &&= (\vec{V},((\texttt{dir+n})\%4),\texttt{cc})\\
&\text{calc}(\texttt{add},\texttt{pos},n_1\cdot n_2\cdot\vec{V},\texttt{ptr}) &&= ((n_2+n_1)\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{subtract},\texttt{pos},n_1\cdot n_2\cdot\vec{V},\texttt{ptr}) &&= ((n_2-n_1)\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{divide},\texttt{pos},n_1\cdot n_2\cdot\vec{V},\texttt{ptr}) &&= (\lfloor n_2\div n_1\rfloor\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{multiply},\texttt{pos},n_1\cdot n_2\cdot\vec{V},\texttt{ptr}) &&= ((n_2\times n_1)\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{modulo},\texttt{pos},n_1\cdot n_2\cdot\vec{V},\texttt{ptr}) &&= ((n_2\ \%\ n_1)\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{greater},\texttt{pos},n_1\cdot n_2\cdot\vec{V},\texttt{ptr}) &&= ((n')\cdot\vec{V},\texttt{ptr})\\
&\textit{Where: } n' &&= \begin{cases}1 & \text{if } n_2 > n_1\\ 0 & \text{otherwise}\end{cases}\\
&\text{calc}(\texttt{not},\texttt{pos},n\cdot\vec{V},\texttt{ptr}) &&= ((n')\cdot\vec{V},\texttt{ptr})\\
&\textit{Where: } n' &&= \begin{cases}1 & \text{if } n = 0\\ 0 & \text{otherwise}\end{cases}\\
&\text{calc}(\texttt{duplicate},\texttt{pos},n\cdot\vec{V},\texttt{ptr}) &&= (n\cdot n\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{roll},\texttt{pos},t\cdot d\cdot\vec{V},\texttt{ptr}) &&= (f^{|t|}(\vec{V}),\texttt{ptr})\\
&\textit{Where: } f(n_0\cdot n_1\cdot\ldots\cdot n_{d-1}\cdot n_d\cdot\vec{V}) &&= \begin{cases}n_1\cdot n_2\cdot\ldots\cdot n_d\cdot n_0\cdot\vec{V} & \text{if } t > 0\\ n_d\cdot n_0\cdot\ldots\cdot n_{d-2}\cdot n_{d-1}\cdot\vec{V} & \text{if } t < 0\\ n_0\cdot n_1\cdot\ldots\cdot n_{d-1}\cdot n_d\cdot\vec{V} & \text{if } t = 0\end{cases}\\
&\text{calc}(\texttt{in\_int},\texttt{pos},\vec{V},\texttt{ptr}) &&= (\texttt{to\_int(read\_char())}\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{in\_chr},\texttt{pos},\vec{V},\texttt{ptr}) &&= (\texttt{read\_char()}\cdot\vec{V},\texttt{ptr})\\
&\text{calc}(\texttt{out\_int},\texttt{pos},n\cdot\vec{V},\texttt{ptr}) &&= (\vec{V},\texttt{ptr})\\
&n \textit{ is printed to } \texttt{stdout} &&\\
&\text{calc}(\texttt{out\_chr},\texttt{pos},n\cdot\vec{V},\texttt{ptr}) &&= (\vec{V},\texttt{ptr})\\
&\texttt{to\_chr}(n) \textit{ is printed to } \texttt{stdout} &&
\end{aligned}
$$

NB: $f^2(x)$ is notation for $f(f(x))$

Qualifiers:
The first two definitions deal with having only 0 values or 1 value on the stack. For all functions that do not increase the stack, if the stack stores less than one value, do not perform the operation. Instead, return the passed-in values. For all binary operations, if the stack has less than 2 values, do not perform the operation.

`no_op`
Return the stack and pointer as-is.

`push`
Push the size of the current region onto the stack. This is found using the `search` function defined in Section 3.6.1

`pop`
Remove the top value of the stack.

`switch`
Take the top value on the stack and toggle the codel choice that many times. For example, if `cc` = 0 and $n = 3$, the result would be `cc` = 1.

`rotate`
Take the top value on the stack and rotate the direction pointer that many times. For example, if `dir` = `Up` and $n = 2$, the result would be `dir` = `Down`.

Binary Operations (`add,subtract,divide,multiply,modulo,greater`)
Pop the top two values off of the stack, perform the binary operation, and push the result onto the stack.

`not`
Perform the boolean not operation on the top value of the stack, and push the result onto the stack.

`duplicate`
Push a copy of the top value onto the stack.

`roll`
A single roll either moves the value at the top of the stack to the depth d, or vice versa, depending on the sign of t. Repeat until $|t|$ reaches 0.

`in_int,in_chr`
Take an input from STDIN, and push it onto the stack. Characters are stored as their integer representation.

`out_int,out_chr`
Pop the top value and print it to STDOUT. To print as a character, first convert from the integer representation to the character.

3.6 Auxiliary Functions

The following functions provide important functionality that is used within the semantic rules in Section 3.7. They are meta-level functions used only in the definition of the operational semantics.

3.6.1 Search

When calculating which position to move to next, and for `push`, Piet must know information about the current colour region. This means knowing where the edges are in the region for the next movement, and knowing the size of the region for `push`. There is no standard implementation for this. The sole requirement is the `search` must be able to capture every codel in any 4-directionally contiguous region.

A region, R, can be defined inductively as follows:

$$R_0 = \{\texttt{(r,c)}\},\ k = \texttt{img((r,c))}$$
$$N_0 = \{\texttt{(r',c')} : (\forall\texttt{(r,c)} \in R_0)\ .\ [\texttt{(r} \pm \texttt{size,c)} \lor \texttt{(r,c} \pm \texttt{size)}]\}$$

Every region starts with a single codel, that codel has a colour, k. It has a set of neighbours, N, that are directly adjacent.

$$R_1 = \{\texttt{(r,c)} : (\forall\texttt{(r,c)} \in N_0)\ .\ \texttt{img((r,c))} = k\} \cup R_0$$

R_1 is every neighbour codel in N_0 so long as their colour is also k, as well as the initial codel in R_0. This gives a new neighbours set:

$$N_1 = \{\texttt{(r',c')} : (\forall\texttt{(r,c)} \in R_1)\ .\ [\texttt{(r} \pm \texttt{size,c)} \lor \texttt{(r,c} \pm \texttt{size)}]\}$$

This can be built upon, such that:

$$R_n = \{\texttt{(r,c)} : (\forall\texttt{(r,c)} \in N_{n-1})\ .\ \texttt{img((r,c))} = k\} \cup R_{n-1}$$
$$N_n = \{\texttt{(r',c')} : (\forall\texttt{(r,c)} \in R_n)\ .\ [\texttt{(r} \pm \texttt{size,c)} \lor \texttt{(r,c} \pm \texttt{size)}]\}$$

In theory, this could continue on indefinitely, but a region, R, is found when $|R_n| = |R_{n-1}|$. That is to say a region is found if, after the inductive step, the set containing the region does not grow.

Any search implementation must be able to find all codels in any R that satisfy the above definition.

3.6.2 Increment & MaybeReset

Increment:

$$\texttt{increment(inc,(dir,cc),}k\texttt{)} = \begin{cases} \texttt{(inc+1,((dir+1)\%4,cc))} \\ \text{if } k \neq \texttt{White} \land \texttt{inc\%2 = 0} \\ \\ \texttt{(inc+1,(dir,(cc+1)\%2))} \\ \text{if } k \neq \texttt{White} \land \texttt{inc\%2 = 1} \\ \\ \texttt{(inc+1,(dir+1)\%4,(cc+1)\%2))} \\ \text{if } k = \texttt{White} \end{cases}$$

MaybeReset:

$$\texttt{maybeReset(inc,}k_1\texttt{,}k_2\texttt{)} = \begin{cases} \texttt{(inc)} & \text{if } k_1 = \texttt{White} \land k_2 = \texttt{White} \\ 0 & \text{otherwise} \end{cases}$$

These functions are called during failed and successful movement respectively. `increment` increments the increment pointer and if the current colour is not White, either rotates the direction pointer or toggles the codel choice. If the colour is White, it increments both. `maybeReset` sets the increment pointer back to 0 so long as the current colour and next colour are not both White.

3.6.3 Δ Function

$$\Delta(k_1, k_2) = \begin{cases} (0,0) & \text{if } k_1 = \texttt{White} \vee k_2 = \texttt{White} \\ ([\text{hue}(k_2) - \text{hue}(k_1)]\%\, 6, [\text{shade}(k_2) - \text{shade}(k_1)]\%\, 3) & \text{otherwise} \end{cases}$$

Definition for calculating the Δ value of two colours. While a Δ with White and any other colour results in (0,0), a Δ calculation with Black is not possible.

3.6.4 Movement Functions

Movement is the most complex part of Piet. The `find_next` auxiliary function deals with finding the next codel to move to. This depends on the current colour.

$$\texttt{find\_next(pos,(cc,dir))} = \begin{cases} \texttt{next(pos,dir)} & \text{if } \texttt{img(pos)} = \texttt{White} \\ \texttt{next(colour\_step(pos,(cc,dir)),dir)} & \text{otherwise} \end{cases}$$

If the current codel is White, the `next` function is used. This means the pointer simply attempts to move one codel in the direction of the direction pointer.

$$\texttt{next((r,c),dir,size)} = \begin{cases} \texttt{(r,c+size)} & \text{if } \texttt{dir = Right} \\ \texttt{(r+size,c)} & \text{if } \texttt{dir = Down} \\ \texttt{(r,c+size)} & \text{if } \texttt{dir = Left} \\ \texttt{(r-size,c)} & \text{if } \texttt{dir = Up} \end{cases}$$

With hue-shade movement, this involves finding the codel or codels that are furthest in the direction of the direction pointer. These codels might be disjoint. If there are multiple codels that are joint furthest, the codel choice is used to break the tie. Formally, R represents a set following the definition in Section 3.6.1, set S represents the positions of codels in R that are the furthest in one of the four directions.

$$S = \begin{cases} \{\texttt{(r,c')} \in R : (\forall \texttt{(r,c)} \in R)\ \texttt{c'} \geq \texttt{c}\} & \text{if } \texttt{dir=Right} \\ \{\texttt{(r',c)} \in R : (\forall \texttt{(r,c)} \in R)\ \texttt{r'} \geq \texttt{r}\} & \text{if } \texttt{dir=Down} \\ \{\texttt{(r,c')} \in R : (\forall \texttt{(r,c)} \in R)\ \texttt{c'} \leq \texttt{c}\} & \text{if } \texttt{dir=Left} \\ \{\texttt{(r',c)} \in R : (\forall \texttt{(r,c)} \in R)\ \texttt{r'} \leq \texttt{r}\} & \text{if } \texttt{dir=Up} \end{cases}$$

S could have a cardinality of 1 or of more than 1. If $|S| = 1$, return the value in it. Otherwise, use the codel choice to break the tie. `tiebreak`, given a direction pointer and codel choice calculates this.

$$\texttt{colour\_step(pos,(cc,dir))} = \begin{cases} S & \text{if } |S| = 1 \\ \texttt{tiebreak(}S\texttt{,(dir,cc))} & \text{otherwise} \end{cases}$$

`tiebreak` finds the codel in S that is furthest in the direction of the codel choice. As a reminder, codel choice represents the relative left and right of the direction pointer in the direction it faces. `cc` = 0 represents the relative left, `cc` = 1 represents the relative right. This is why there are only 4 possibilities, as a leftward turn from left and a rightward turn from right would face down, for example. Thus, the first possibility finds the codel with the largest row component. This is repeated for the other 3 possibilities.

$$\texttt{tiebreak}(S,(\texttt{dir},\texttt{cc})) = \begin{cases} (\texttt{r'},\texttt{c}) : (\forall \texttt{r} \in S)\ \texttt{r'} > \texttt{r} & \text{if}\,(\texttt{dir},\texttt{cc}) = (\texttt{Right},\texttt{1}) \vee (\texttt{dir},\texttt{cc}) = (\texttt{Left},\texttt{0}) \\ (\texttt{r},\texttt{c'}) : (\forall \texttt{c} \in S)\ \texttt{c'} > \texttt{c} & \text{if}\,(\texttt{dir},\texttt{cc}) = (\texttt{Down},\texttt{0}) \vee (\texttt{dir},\texttt{cc}) = (\texttt{Up},\texttt{1}) \\ (\texttt{r'},\texttt{c}) : (\forall \texttt{r} \in S)\ \texttt{r'} < \texttt{r} & \text{if}\,(\texttt{dir},\texttt{cc}) = (\texttt{Right},\texttt{0}) \vee (\texttt{dir},\texttt{cc}) = (\texttt{Left},\texttt{1}) \\ (\texttt{r},\texttt{c'}) : (\forall \texttt{c} \in S)\ \texttt{c'} < \texttt{c} & \text{if}\,(\texttt{dir},\texttt{cc}) = (\texttt{Down},\texttt{1}) \vee (\texttt{dir},\texttt{cc}) = (\texttt{Up},\texttt{0}) \end{cases}$$

Once the tie is broken, `next` is called. As this has found the edge codel in the current region, an additional step needs to be taken to attempt to move the pointer into the next region. So, the pointer, having moved to the edge codel, tries to move one codel in the direction of the direction pointer. This uses the `next` function defined above.

3.7 State Transitions

Piet can be in two states. This paper calls them IDLE and EVAL. Initially, execution starts in the IDLE state.

So, the initial state vector $\mathcal{S}_0$ is as follows:

$$\mathcal{S}_0 = \{\texttt{img},\texttt{size},\epsilon,(\texttt{Right},\texttt{0}),(\texttt{0},\texttt{0}),\texttt{0}\}$$

3.7.1 IDLE State

IDLE has a single condition, represented by the two rules. IDLE checks if the increment pointer is less than 9, that is to say 8 consecutive movement attempts to correct execution have not occurred. IDLE-success represents increment pointer being less than 9, which has the state transition to the EVAL state. IDLE-failure occurs when the inc is equal to (or greater than) 9, in which case execution halts.

$$\boxed{\mathcal{S} \longrightarrow \mathcal{S}'}$$

$$\text{IDLE-success}\ \frac{\texttt{inc} < 9}{\langle \text{IDLE}, \overrightarrow{V}, (\texttt{pos},\texttt{pointer},\texttt{inc})\rangle \longrightarrow \langle \text{EVAL}, \overrightarrow{V}, (\texttt{pos},\texttt{pointer},\texttt{inc})\rangle}$$

$$\text{IDLE-failure}\ \frac{\texttt{inc} \geq 9}{\langle \text{IDLE}, \overrightarrow{V}, (\texttt{pos},\texttt{pointer},\texttt{inc})\rangle \longrightarrow F}$$

3.7.2 EVAL State

EVAL also has two rules. Both start in the same manner. Using the position, find out the colour of the current position in the image array (k). Then, using the find_next auxiliary function, attempt to find the next possible position and calculate its colour (k'). This is where the rules split.

$$\boxed{\mathcal{S} \longrightarrow \mathcal{S}'}$$

$$\text{EVAL-success}\ \frac{\begin{array}{c} \texttt{img(pos)=}k \quad \texttt{find\_next(pos,ptr)=pos'} \quad \texttt{img(pos')=}k' \\ k' \neq \texttt{Black} \qquad\qquad \Delta(k,k') = \texttt{op} \\ \textbf{calc}(\texttt{op},\texttt{pos},\overrightarrow{V},\texttt{ptr}) = (\overrightarrow{V'},\texttt{ptr}') \quad \texttt{maybeReset}(\texttt{inc},k,k') = \texttt{inc}' \end{array}}{\langle \text{EVAL}, \overrightarrow{V}, (\texttt{pos},\texttt{ptr},\texttt{inc})\rangle \xrightarrow{\texttt{img,size}} \langle \text{IDLE}, \overrightarrow{V'}, (\texttt{pos'},\texttt{ptr'},\texttt{inc'})\rangle}$$

$$\textsc{eval}\text{-failure}\ \frac{\texttt{img(pos)=}k \quad \texttt{find\_next(pos,ptr)=pos'} \quad \texttt{img(pos')=}k' \qquad k' = \texttt{Black} \quad \texttt{increment(pos,ptr,inc,}k\texttt{)} = \texttt{(pos,ptr',inc')}}{\langle \textsc{eval}, \overrightarrow{V}, \texttt{(pos,ptr,inc)}\rangle \xrightarrow{\texttt{img,size}} \langle \textsc{idle}, \overrightarrow{V}, \texttt{(pos,ptr',inc')}\rangle}$$

Following EVAL-success, if k' is not Black, calculate the Δ of k and k'. This provides an operation, based on the Mapping in Table 3.2. This operation is called with the **calc** auxiliary function, which also takes position, pointer, and the stack. **calc** returns the new values for the pointer, and stack. Finally, call the `maybeReset` function to modify the increment pointer value, and return to the IDLE State with the new position, stack, pointer and increment values.

EVAL-failure, on the other hand, occurs when k' is Black. If so, no Δ is calculated, no operation is executed and no new position taken. The `increment` function is called which increases the increment pointer, and modifies the direction pointer and codel choice. Once this all occurs, execution returns to the IDLE state, with its unchanged position and stack, modified direction pointer and codel choice values and increased increment pointer.

4 Implementation

4.1 The *Magpie* Interpreter

Design of the rules went hand-in-hand with the creation of the interpreter. This allowed for an iterative process of creation where, taking the specification, rudimentary functions were created. Then, after having tested the functions, the rules and definitions were refined down to a purer form for the formalisation. This then fed back into the code, where the new definitions were tested to ensure proper execution.

Alongside the operational semantics, *Magpie* was created. This is a reference interpreter written in OCaml, with the name a reference to *piet* being a regional English term for the bird. OCaml has a history as a language used to implement interpreters and compilers for other programming languages, including for Rust as it was being developed (Baturin 2024). OCaml, as a functional programming language, approaches the style of the operational semantics. For example, OCaml has support for first-class functions as would be necessary in the calc function. In order to evaluate the operational semantics, *Magpie* will be compared against other Piet interpreters.

4.2 Development

The decision was taken to split the *Magpie* into several modules, each dealing with a certain subsection of interpreter functions. These are split based on similarity.

4.2.1 Colour

The first step before running the code is turning a .png or .gif image into the internal representation needed for execution. Using the CamlImages library, the function runs pixel-by-pixel converting each into a hue-shade or a tone, defaulting to Black if the RGB value falls outside of the mapping. Due to issues with the library, although CamlImages lists several optional dependencies, these are all mandatory dependencies for *Magpie* to correctly function.

With the image array created, the main function instantiates the state record, with the values as in S_0. Execution then moves between the idle function, representing the IDLE rules, and the eval function representing the EVAL rules.

4.2.2 Operations

Pietops.ml contains a function for each of the 18 specification operations. There is an additional implementation-specific operation. This is to follow the normal access rules of a stack abstract data type. Two pop functions exist. `pop`, which implements the pop operation in the Piet specification, which does not return a value, and `popv`. `popv` returns the integer value removed from the top of the stack, and is used in the other operation functions when operating over the stack. It also contains the op function to turn a delta value into a function call. While the semantics describe the high-level result after one roll, in order to conserve the stack implementation, additional recursive functions were created to pop the intermediate section between the head and value at depth *d*.

Magpie does not implement automatic codel sizing. This means when code is run at an incorrect size, the integer values representing characters would be a different value to what they should be. Furthermore, OCaml does not implement complete integer to Unicode character conversion. This means full Unicode support in `out_char` is not possible. These issues both compounded and caused the interpreter to halt. *Magpie* instead prints a replacement character, "?". This allows execution to continue, while potentially representing to a user that this code may be running incorrectly.

4.2.3 Movement

The `search` function is implemented using a modified flood-fill algorithm. Flood-fill or seed-fill is a graphics algorithm used to alter the colour of pixels in a given region (Foley 1995, pp. 980–981). Figure 4.1 presents the implementation in *Magpie*.

```
let rec search (img : Colour.colour array array) (i, j) (size: int) stack
    (colour: Colour.colour) =
if i < 0 || j < 0 || i >= Array.length img || j >= Array.length img.(0) then
  stack
else if img.(i).(j) <> colour then stack
else if List.mem (i, j) stack then stack
else
  let newstack = (i, j) :: stack in
  let nodes = search img (i, j + size) size newstack colour in
  let nodes_1 = search img (i + size, j) size nodes colour in
  let nodes_2 = search img (i, j - size) size nodes_1 colour in
  let nodes_3 = search img (i - size, j) size nodes_2 colour in
  nodes_3
```

Figure 4.1: `search` *implementation in Magpie, using flood-fill.*

`gather_coords` is a function that finds the set *S* as defined above, and provides the same high-level result as described in Section 3.6.4, but uses filters and mapping to create the *S* set. This seems acceptable as it is expected implementations will differ on the internals but will conform to the semantics. `gather_coords` sorts all the codels in the region based either on their row or column component, depending on the direction of the direction pointer. Then, following the set-builder rules in Section 3.6.4, `gather_coords` finds the codels that satisfy the rule to create *S*.

During the testing of *Magpie*, certain Piet images did not run correctly . It was unclear, from the outset, why this was happening. Later, it transpired that OCaml's `fprintf` functions did not flush `stdout`. This meant that code would run, but never print an output if the code was non-terminating. This came in tandem with another issue which was failed movement on White. The online specification is, again, vague with the definition of failed movement. But specifically, it seems, when movement is impossible on a White codel, both the direction pointer and codel choice should be updated. After testing this in *Magpie*, this was fed back into the semantics.

```
let eval state =
 let row, col = !(state.pos) in
 let c1 = state.pic.image.(row).(col) in
 let nextRow, nextCol = find_next(state) in
  if Bool.not (checkstep state (nextRow,nextCol)) then ( (*eval-success:*)
   let c2 = state.pic.image.(nextRow).(nextCol) in
   let d = delta c1 c2 in
   if !verbose then (
    Printf.printf "Delta: %d,%d\n" (fst d) (snd d);
    Printf.printf "Operation: %s\n" (string_of_delta d))
   else ();
   try (*Catches any errors that may arise.*)
    mapfuncs state d;
    state.pos := (nextRow, nextCol);
    state.inc := maybeReset state.inc c1 c2;
    printStep (nextRow, nextCol, c2)
   with Failure _ -> (*Treats expression as no_op if error occurs.*)
    state.pos := (nextRow, nextCol);
    state.inc := maybeReset state.inc c1 c2;
    printStep (nextRow, nextCol, c2))
  else ( (*eval-failure:*)
   if !verbose then
    Printf.printf "Position %d, %d inaccessible. Incrementing Pointer."
     nextRow nextCol
   else ();
  increment state c1)
```

(a) *EVAL function implemented in Magpie.*

$\mathcal{S} \longrightarrow \mathcal{S}'$

$$\text{Eval-success}\ \frac{\begin{array}{c}\texttt{img(pos)=}k \quad \texttt{find\_next(pos,ptr)=pos'} \quad \texttt{img(pos')=}k' \\ k' \neq \texttt{Black} \qquad \Delta(k,k') = \texttt{op} \\ \textbf{calc}(\texttt{op},\texttt{pos},\overrightarrow{V},\texttt{ptr}) = (\overrightarrow{V'},\texttt{ptr}') \quad \texttt{maybeReset}(\texttt{inc},k,k') = \texttt{inc}'\end{array}}{\langle \textsc{eval}, \overrightarrow{V}, (\texttt{pos,ptr,inc})\rangle \xrightarrow{\texttt{img,size}} \langle \textsc{idle}, \overrightarrow{V'}, (\texttt{pos',ptr',inc'})\rangle}$$

$$\text{Eval-failure}\ \frac{\begin{array}{c}\texttt{img(pos)=}k \quad \texttt{find\_next(pos,ptr)=pos'} \quad \texttt{img(pos')=}k' \\ k' = \texttt{Black} \quad \texttt{increment(pos,ptr,inc,}k\texttt{) = (pos,ptr',inc')}\end{array}}{\langle \textsc{eval}, \overrightarrow{V}, (\texttt{pos,ptr,inc})\rangle \xrightarrow{\texttt{img,size}} \langle \textsc{idle}, \overrightarrow{V}, (\texttt{pos,ptr',inc'})\rangle}$$

(b) *A reminder of the EVAL rules.*

Figure 4.2: *EVAL code from Magpie, and a reminder of the EVAL semantic rules.*

4.2.4 Semantics

As mentioned above, IDLE and EVAL are implemented as functions in *Magpie*. Figure 4.2a is a code snippet from *Magpie*, showing the implementation of the semantic rules. Parts of the code implement quality of life improvements detailed in Section 4.2.5, for example, verbose execution (`!verbose`).

This function implements both rules dealing with the EVAL state. `checkstep` checks whether the colour of the next position is Black, or outside the range of the image. This is implicit in the rules that execution cannot step outside of the image boundaries, but this needs to be checked in the interpreter.

The first if statement splits the function into the Success and Failure rules. There are small differences to help map the semantics into OCaml. These changes are not to the semantics' detriment, however, as they do not materially diverge from the high-level semantics.

4.2.5 Quality Of Life

The interpreter implements a command line parser to take the image file, codel size and execution options, including stepwise and verbose execution. This is important as it allows for Piet code to be tested using *Magpie* with it providing step by step updates to the state and the current Δ value and next operation.

5 Evaluation

5.1 Evaluation Set

As there is no standard interpreter, *Magpie* was tested against a set of Piet interpreters taken from the Piet website (Morgan-Mar 2023*b*). This set will be referred to as the evaluation set. These are:

1. *npiet*, by Erik Schoenfelder, written in C (Schoenfelder n.d.),
2. Frank Zago's C interpreter (Zago 2020),
3. Jens Bouman's Python interpreter (Bouman 2024),
4. *rpiet*, by Philipp Tessenow, written in Rust (Tessenow 2024),
5. Ynn's Rust interpreter (ynn 2024).

The evaluation set represents five implementations in three languages. This provides a good base to compare and contrast *Magpie*'s results with pre-existing interpreters. This does, however, cause an issue of there not being a "standard" result. This evaluation, instead, relies on an expected result, given a description of the code. Code, however, may be malformed or faulty, and would therefore not return the expected result. Success, in this case, would mean *Magpie*'s output agreeing with the output of the interpreters in the evaluation set. That is to say, if a Piet code isn't properly written, the evaluation set will fail. *Magpie* should fail in that same way.

Some interpreters were modified to update deprecated libraries to modern versions or to silence warnings. This, however, was limited to exclusively modifying `import` or `#include` calls or adding a silence warnings macro. No material modifications to code were made.

5.2 Method

A Python script was written which, given an image, could run each interpreter in turn as a subprocess and collate their outputs. Each subprocess is given 25 seconds to run, 100 if the code required input, otherwise the subprocess is killed. When this occurs, or the interpreter panics, no result is returned. Instead of its output, a string stating the fault is returned. This is an issue for code that is non-terminating by design. Instead, these were run by hand.

5.3 Discussion of Results

A selection of code and results is provided in this chapter, with images in Appendix B and a list of results not detailed here in Appendix C. The choice in the table was limited by the need to represent the expected output and the outputs in a given space. More text-heavy examples are given following Table 5.1.

Table 5.1: *A selection of Piet code results. A green box with no value means the interpreter returned the expected output. Red represents an output that was not the expected output, with the output given. A yellow box represents a time out (T.O), or a small difference in output. A brown box with a P means the interpreter panicked.*

	Input	*Expected Output*	**Magpie**	1 *npiet*	2 *Zago*	3 *Bouman*	4 *rpiet*	5 *Ynn*
piet_factorial	6	720	🟩	🟩	🟩	🟩	🟩	🟫 *P*
piet_factorial	15	*1.308e+12*	🟩	🟥 2.0e+9	🟥 2.0e+9	🟩	🟩	🟫 *P*
power_2	2 5	32	🟩	🟩	🟩	🟥 3	🟩	🟩
power_2	10 10	*10^10*	🟩	🟥 1.4e+9	🟥 1.4e+9	🟥 3	🟥 310	🟩
Countdown		*[10 - 1 on separate lines]*	🟩	🟩	🟩	🟩	🟩	🟨 *Double -spaced*
alphafilledbig		*[a-z]*	🟩	🟩	🟩	🟨 *T.O*	🟨 *T.O*	🟩
dayofweek	2025 3 28	5 *(Friday)*	🟩	🟩	🟩	🟫 *P*	🟨 *T.O*	🟩
euclid_clint	14 9	1	🟥 4	🟥 4	🟥 4	🟫 *P*	🟥 4	🟥 4
DivideByZero		*N/A*	0	0	0	*E.T*	0	0
piet_pi		31405	🟩	🟩	🟩	🟩	🟩	🟩
piet_pi_big (size = 1)		*integer approximation of pi*	🟨 *T.O*	🟥 0Z	🟥 0Z	🟨 *T.O*	🟥 2Z	🟥 2 Z
piet_bfi_16	,+>,+>,+> ,+.<.<.<. lsdhO\n	Piet	🟩	🟩	🟨 *T.O*	🟨 *T.O*	🟩	🟩
primetest2	6	*not prime*	🟥 6isprime					

5.3.1 Table of Selected Results

Table 5.1 highlights that *Magpie* provides the expected output to a given image in most circumstances. Also, where at least one interpreter in the evaluation set returned the expected output, *Magpie* did as well. Where a majority of the evaluation set produced an incorrect output, *Magpie* returned that same incorrect output. This implies that *euclid_clint*, code to execute Euclid's algorithm, and *primetest2*, a prime test, are both faulty code.

DivideByZero is an interesting case, in that the online specification provides no details on a division by zero, making it implementation dependent. Therefore, any plausible outcome is acceptable.

Looking at *npiet* and Zago's interpreter for *piet_factorial* and *power_2*, both fail in the same way. This is not due to poor implementation, but due to the maximum value an int can hold in C. 15! and 10^{10} are larger than $2^{31} - 1$. This is likely the same for *rpiet*.

It is likely Bouman's Python interpreter implements the Piet specification incorrectly, as it provides the expected or a consensus output under half of the time.

piet_pi uses a circle and its radius to calculate an approximation for pi. All interpreters are able to return the expected output. *piet_pi_big* is the same image, with a codel size of 3. This implies that a codel size of 1 could be attempted, giving a more accurate approximation. This was not the case. All code failed to produce the expected output, likely meaning the assumption is incorrect. It is unclear what the 2Z or 0Z necessarily refers to. *Magpie* in this instance, timed out. This is due to the need to calculate the region size of the circle. Flood-fill is a recursive algorithm that will revisit codels multiple times. This is the method chosen in implementing the semantics for *Magpie*, which clearly has a detriment on the runtime. It does not necessarily imply, however, that flood-fill is wrong to be used. Flood-fill looks to be a good base for an implementation, but countless improvements on a simple graphics algorithm are sure to exist.

5.3.2 Quine

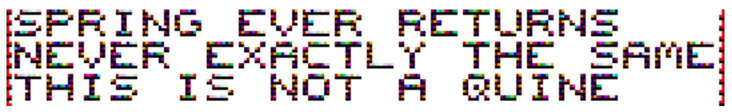

Figure 5.1: *A Piet quine, which prints the haiku shown, by Kelly Boothby (Morgan-Mar 2023*a*).*

A quine is code that prints out itself. Figure 5.1 is an example Piet code. This code is non-terminating and as such had to be run by hand.

The expected output, therefore, is the following:

```
SPRING EVER RETURNS
NEVER EXACTLY THE SAME
THIS IS NOT A QUINE
```

Magpie, npiet, Zago's interpreter and Ynn's interpreter provided the expected output and did not terminate. Bouman's interpreter provided the following, before panicking:

```
ŌňĤĜĔŘĔňňĔŐŔňŌ(ĔŘĔňĔŠĄČŐİŤŐĠĔŌĄĴĔ(ŐĠĤŌĤŌļŐĄńŔĤĔ
```

rpiet did not print anything, nor did it terminate.

5.3.3 Game of Life

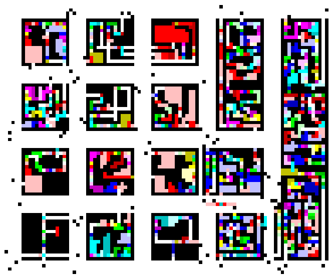

Figure 5.2: *Piet code which runs an interactive Game of Life simulation, by Geerten Vink (Morgan-Mar 2023*a*).*

Game of Life is the famous cellular automaton created by mathematician John Conway (Gardner 1970). Figure 5.2 is Piet code for an interactive step-by-step simulation of this automaton. Giving each interpreter the same input, provided in Appendix C, the interpreter's should end at the state shown in Figure 5.3. npiet, Zago's interpreter, Ynn's interpreter and Magpie correctly produce that state. Bouman's interpreter panicked, *rpiet* timed out.

5.4 Evaluation Analysis

Reflecting on the results discussed, *Magpie* performs as well as interpreters in the evaluation set. In addition, where code failed in other interpreters due to poor Piet code, *Magpie* failed in the

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . o .
. . . . . . . o . o
. . . . . . . o . o
. . . . . . . . o .
```

Figure 5.3: *Expected output from the Game of Life Piet code.*

same way. In certain cases, *Magpie* performed better due to OCaml's larger integer value size. When running certain large code, however, the flood-fill algorithm becomes a slight detriment to performance, leading to slower execution times, and execution that overran the subprocess timer. In addition, interpreters, such as Jens Bouman's Piet interpreter look to have implemented parts of the Piet specification incorrectly, leading to undefined behaviour.

This provides confidence in the accuracy of the formalisation of Piet, and the necessity of a better, more unambiguous collection of definitions and rules. But, implementation specific issues with *Magpie* could be improved, including a better search function.

6 | Conclusion

6.1 Summary

The goal of this project was to create a formal specification for the Piet language. This was due to vague, imprecise and loose definitions in the current specification found online. This meant the project could be split into two parts. Firstly, the creation of the operational semantics. Secondly, using the semantics to create a reference interpreter. This interpreter was used against other Piet interpreters to evaluate the effectiveness of the reference interpreter. The *Magpie* interpreter was the result of this work.

Evaluating *Magpie* against five other Piet interpreters shows that it executes Piet code in the same manner as those created without the operational semantics. This is a good indicator that the operational semantics are a correct formalisation of the current specification. *Magpie* has some issues regarding runtime speed with larger images, but this does not impact the correctness of its results.

6.2 Future Work

On the interpreter side, work can be done to implement automatic codel sizing and make improvements to the flood-fill implementation. This would have the effect of speeding up code execution in certain circumstances. The quality of life of the interpreter could also be improved by reworking the stepwise execution version of the interpreter. This would improve legibility and help debugging of code.

Magpie is not an IDE. But, it would be possible implement a graphical interface for the interpreter, including presenting the user with the pointer's current location on the actual image. In addition, providing programmers with a graphical creation interface for Piet code could also prove fruitful.

On the semantics side, exploring creating a type system for Piet code would allow for modification of the semantic rules, with a more clear differentiation between hue-shade colours and the tones. Additionally, there is space to extend the Piet language, potentially by adding a value store that is outside of the stack. More colours could be added, and a shift to a delta function working over change in red, green, and blue components would potentially add room for more operations. This could be fed into a new interpreter for this language extension.

Though this paper sets up the semantics, due to the project's 20-credit scope, there is no discussion about the limitations of Piet or a deeper analysis of what these rules mean for code creation. For example, what is the smallest possible Piet code that, without taking an input, could print "Hello, World!"? Is there a way to prove that this is the case? These questions could be asked about any possible code. This is left as an exercise to the reader.

6.3 Reflection

This project has been incredibly rewarding. I have learnt much more about the world of esoteric languages in particular, and programming language techniques in general. OCaml, while occasionally very fiddly, is an expressive language that, to me, seems designed for projects like these. It has been interesting to me to see the expressiveness of a language that, at first glance, can look like a disparate collection of colours. As well, the structure of operational semantics is much more fluid and expressive than it may seem initially. This project has made me consider what sort of possible programming language idioms could be conceivable outside of the linear construction of so many languages. The door is still open for a 3-dimensional programming language.

Bibliography

Baturin, D. (2024), 'Preface'.
URL: *https://ocamlbook.org/preface/*

Bouman, J. (2024), 'Piet_interpreter'.
URL: *https://github.com/JensBouman/Piet_interpreter*

Foley, J. D., ed. (1995), *Computer graphics: principles and practice*, Addison-Wesley systems programming series, 2nd ed. in c edn, Addison-Wesley, Reading, Mass.

Gardner, M. (1970), 'Mathematical Games', *Scientific American* **223**(4), 120–123.
URL: *https://www.jstor.org/stable/24927642*

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J. (2017), Bringing the web up to speed with WebAssembly, *in* 'Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation', ACM, Barcelona Spain, pp. 185–200.
URL: *https://dl.acm.org/doi/10.1145/3062341.3062363*

Lewis, D. (1957), *Mondrian (1872 - 1944)*, The Faber Gallery, Faber & Faber, London.

Lyon, J. and Woods, D. (1973), 'The INTERCAL Programming Language Reference Manual'.
URL: *https://3e8.org/pub/intercal.pdf*

Mateas, M. (2008), Weird Languages, *in* M. Fuller, ed., 'Software Studies: A Lexicon', Leonardo, MIT Press, Cambridge, Mass, pp. 267 – 277.

Mondrian, P. (1921), 'Composition with Large Red Plane, Yellow, Black, Grey and Blue'.
URL: *https://rkd.nl/images/218084*

Morgan-Mar, D. (2018), 'DM's Esoteric Programming Languages - Piet'.
URL: *https://www.dangermouse.net/esoteric/piet.html*

Morgan-Mar, D. (2022), 'DM's Esoteric Programming Languages - Chef'.
URL: *https://www.dangermouse.net/esoteric/chef.html*

Morgan-Mar, D. (2023*a*), 'DM's Esoteric Programming Languages - Piet Samples'.
URL: *https://www.dangermouse.net/esoteric/piet/samples.html*

Morgan-Mar, D. (2023*b*), 'DM's Esoteric Programming Languages - Piet Samples [sic]'.
URL: *https://www.dangermouse.net/esoteric/piet/tools.html*

Olmstead, B. (1998), 'Malbolge Specification'.
URL: *http://www.lscheffer.com/malbolge_spec.html*

Pierce, B. C. (2002), *Types and programming languages*, MIT Press, Cambridge, Mass.

Plotkin, G. D. (2004), 'A structural approach to operational semantics', *The Journal of Logic and Algebraic Programming* **60-61**, 17–139.
URL: *https://linkinghub.elsevier.com/retrieve/pii/S1567832604000402*

Programming Languages Software Award (n.d.).
URL: *https://www.sigplan.org/Awards/Software/*

Schoenfelder, E. (n.d.), 'npiet'.
URL: *https://www.bertnase.de/npiet/*

Tessenow, P. (2024), 'rpiet'. original-date: 2019-10-25T10:17:30Z.
URL: *https://github.com/tessi/rpiet*

The GNU C Reference Manual (n.d.).
URL: *https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html*

Turner, J., ed. (1996), *The dictionary of art*, Vol. 22, Grove, New York.

WebAssembly Specification (2025).
URL: *https://webassembly.github.io/spec/core/_download/WebAssembly.pdf*

Wong, C. (2019), A Formal Semantics for Befunge, *in* 'A Record of the Proceedings of SIGBOVIK 2019', Pittsburgh, Penn., pp. 138–143.
URL: *https://sigbovik.org/2019/proceedings.pdf*

ynn (2024), 'piet_programming_language'. original-date: 2023-03-27T13:38:49Z.
URL: *https://github.com/your-diary/piet_programming_language*

Zago, F. (2020), 'piet'. original-date: 2015-06-27T18:49:44Z.
URL: *https://github.com/frank-zago/piet*

A README

```
# Files
The files in this zip are as follows:
- ‘Magpie‘ - Folder storing the *Magpie* Interpreter
- ‘tableau.py‘ - The python evaluation script
- ‘imgs‘ - a collection of test images

# The *Magpie* Interpreter
Magpie is an interpreter for the Piet language, created by David Morgan-Mar.


> [!Danger] Errors or Problems!
> If you run into an error or a problem, you can contact me at:
> 2562340a@student.gla.ac.uk
> or kwieto.alchimowicz@gmail.com

## Running Magpie
Magpie can be run on the command line with dune:
‘dune exec main -- [args]‘

### Arguments
The arguments are as follows
‘dune exec main -- -f <file> -c <codelsize> [-v] [-s]‘
- ‘-f‘ is the Piet image
- ‘-c‘ is the codel size
- ‘-v‘ enables verbose execution
- ‘-s‘ enables stepwise execution

## Requirements
Magpie is written in OCaml 5 and requires the following OCaml libraries to run:
- ‘dune‘
- ‘base‘
- ‘camlimages‘
- ‘ppx_deriving‘
- ‘lablgtk‘
- ‘graphics‘
- ‘conf-libpng‘
- ‘conf-libgif‘
- ‘conf-freetype‘
- ‘conf-ghostscript‘
- ‘odoc‘

## Inputs
*Magpie* may ask for user input when running code. There is nothing printed to
    the terminal when this occurs, so users should be aware when a new line is
    started. A new line will signify requiring input.
```

```
---

# ‘tableau.py‘
‘tableau‘ is a python (3.13.1) script designed to evaluate Piet interpreters.
‘tableau‘ requires a text file (‘interpreters.txt‘) of the interpreters set up in
    the following way:
‘name, ./path/to/interpreter -f %f -c %c‘
‘%f‘ and ‘%q‘ will be replaced by the file path and codel size when evaluating
    the interpreter. E.g
‘‘‘
npiet, ./interpreters/npiet-1.3f/npiet -q %f
C Interpreter,./interpreters/zago-piet/piet %f %c
‘‘‘


## Running ‘tableau‘
 ‘tableau‘ can be run on the command line with ‘python‘:
‘python tableau.py [args]‘

‘tableau‘ asks for input and a codel size when in test mode. ‘tableau‘ uses an
    input file when in individual code evaluation mode.


> [!Warning] ‘tableau‘ and Windows
> ‘tableau‘ runs on Mac and Linux. Due to it using the subprocess module in
    Python, it is unclear whether it will run on Windows machines.

##### Results
‘tableau‘ provides a output to stdout of the results of running the interpreters.
    Identical results from different interpreters are printed once, with the
    interpreters providing that result listed underneath.

### Arguments
The arguments are as follows
‘python tableau.py [-h] [-f FILE] [-i INPUT] [-c CODEL] [-t TEST]‘
  - ‘-h, --help‘ Show the help message
  - ‘-f, --file FILE‘ The file path for the Piet image
  - ‘-i, --input INPUT‘ The file path for a text input
  - ‘-c, --codel CODEL‘ set the codel size

  Alternatively,
  - -t, --test TEST Provide a folder location to run all the code inside the
      folder.


### Img
The ‘img‘ folder has three test images. These are detailed in the README.md found
    in the folder.
```

B Evaluation Code

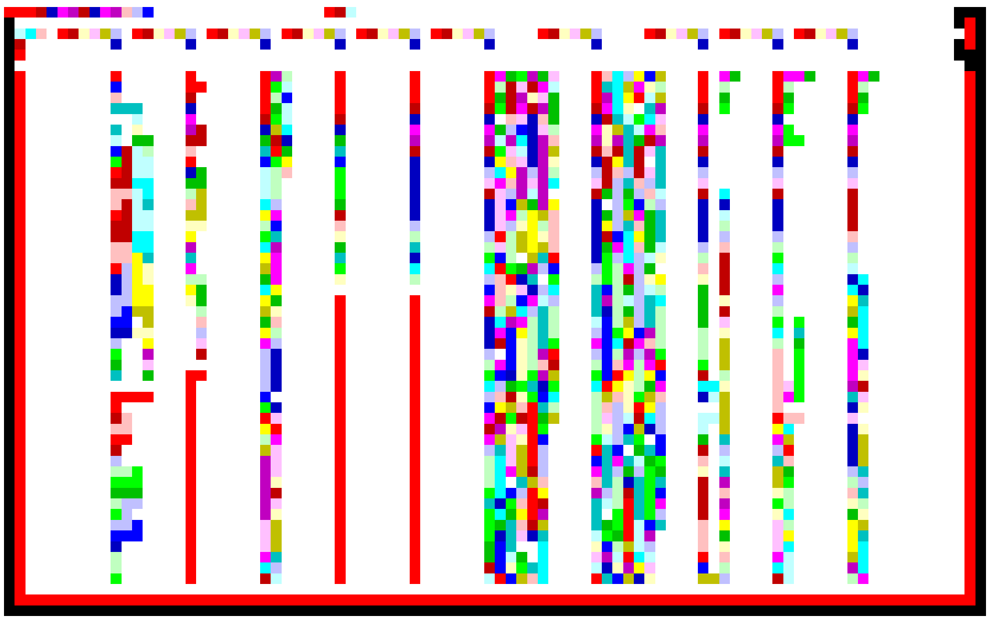

Figure B.1: *99 Bottles – code by Eddy Ferreira (Morgan-Mar 2023a).*

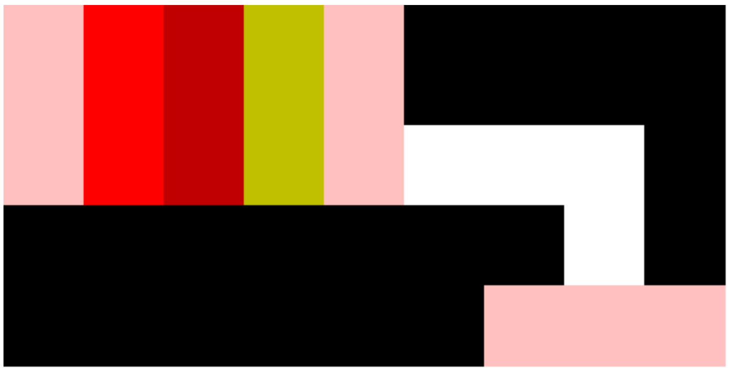

Figure B.2: *Add – Code that calculates 2+2 and prints the result, by Jens Bouman (Bouman 2024).*

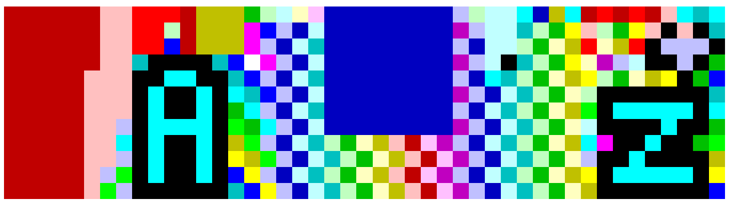

Figure B.3: *alpha_filled_big – code that prints the alphabet, by Chad Etzel(Morgan-Mar 2023a).*

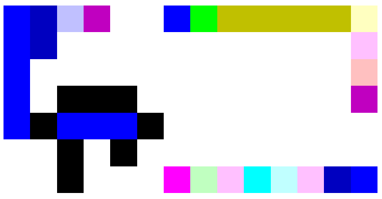

Figure B.4: *Countdown – code that counts from 10 to 1.*

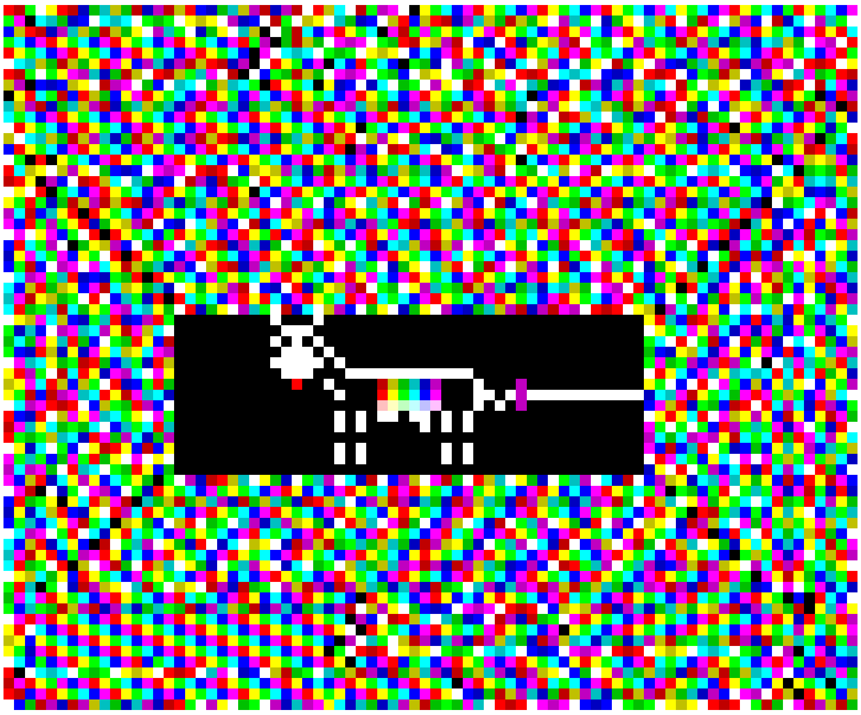

Figure B.5: *Cowsay - code that implements the Cowsay perl script, by Antoine Lucas (Morgan-Mar 2023*a*).*

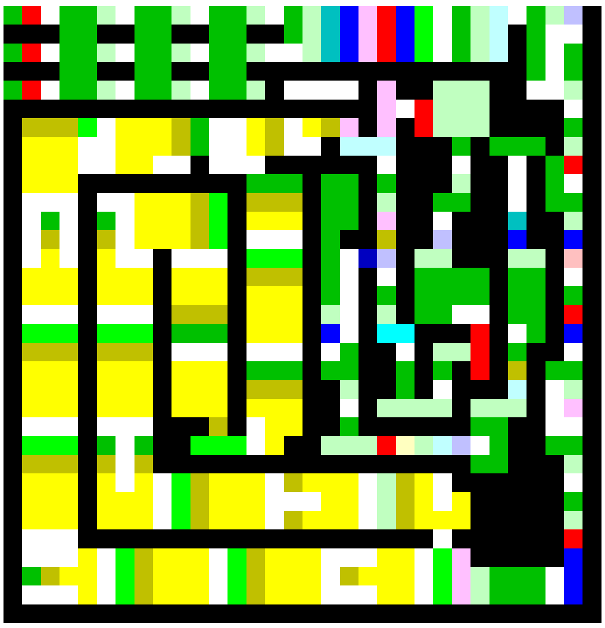

Figure B.6: *dayoftheweek - Given a year, month and day, returns the day of the week, by Jonathan Couper-Smartt (Bouman 2024).*

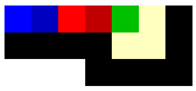

Figure B.7: *DivideByZero – code that attempts to divide by 0, by Jens Bouman (Bouman 2024).*

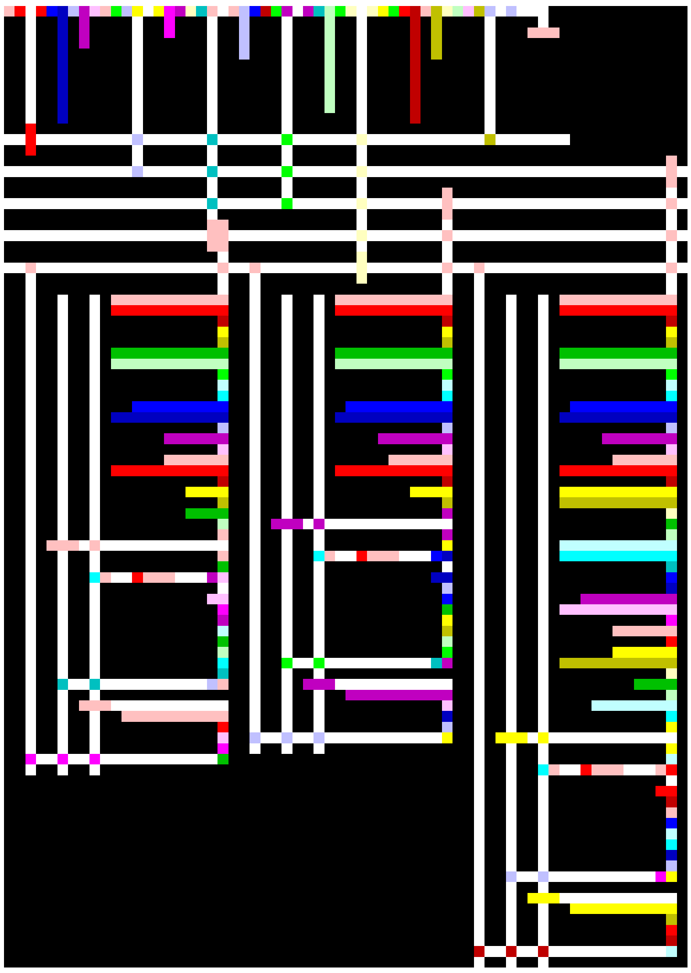

Figure B.8: *Fizzbuzz – code that implements Fizzbuzz up to 20, by Sergei Lewis (Morgan-Mar 2023a).*

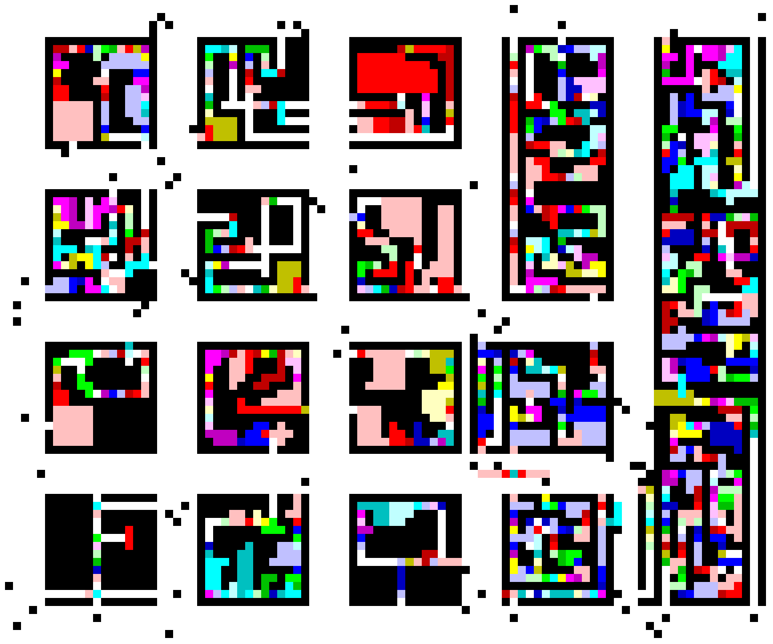

Figure B.9: *GameOfLife - A game of life simulation, by Geerten Vink (Morgan-Mar 2023a).*

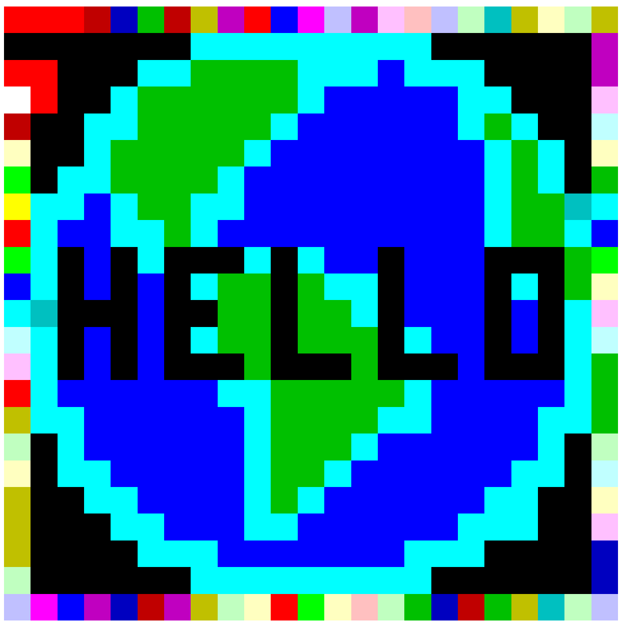

Figure B.10: *helloworldglobe - Code that outputs "Hello, World!" , by Kelly Boothby (Morgan-Mar 2023a).*

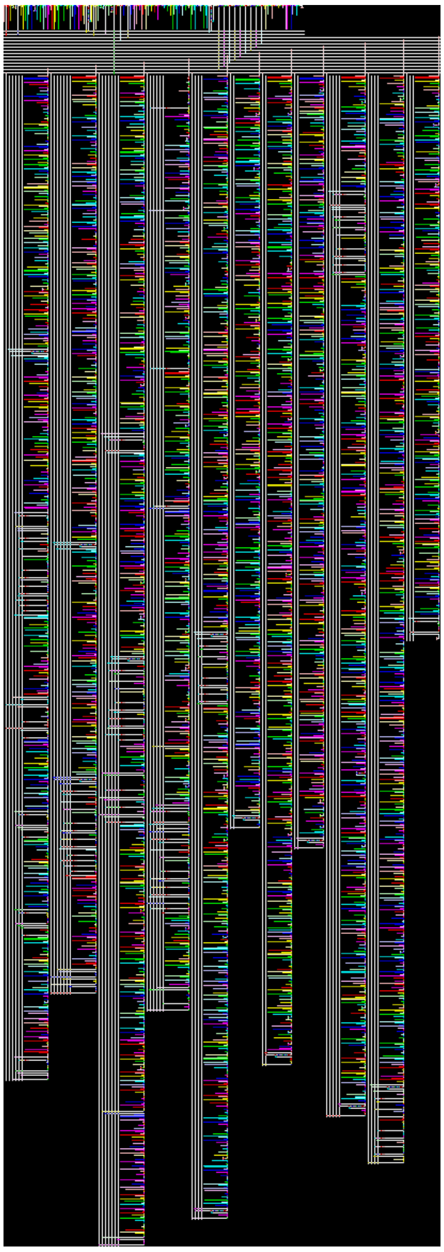

Figure B.11: *pietquest - A text adventure, by Sergei Lewis (Morgan-Mar 2023*a*).*

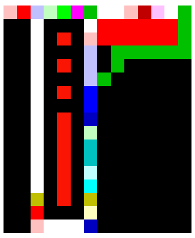

Figure B.12: *piet_factorial - Calculates the factorial of a number, by Thomas Polasek (Morgan-Mar 2023*a*).*

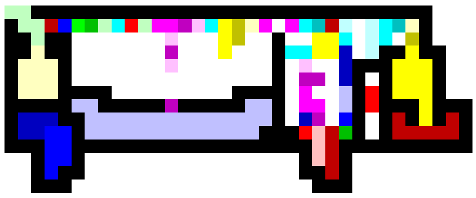

Figure B.13: *power2 - Calculates one number to the power of another, by François Tavin (Morgan-Mar 2023*a*).*

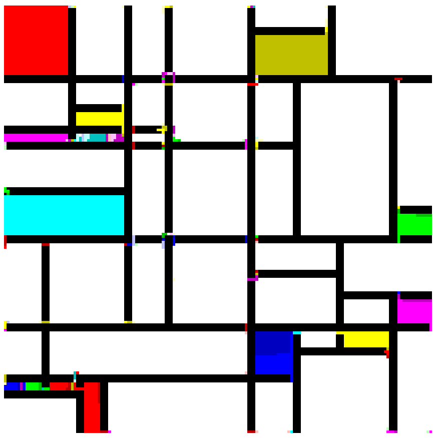

Figure B.14: *primetest2 - Calculates whether a number is prime (likely broken), by Kyle Woodward (Morgan-Mar 2023*a*).*

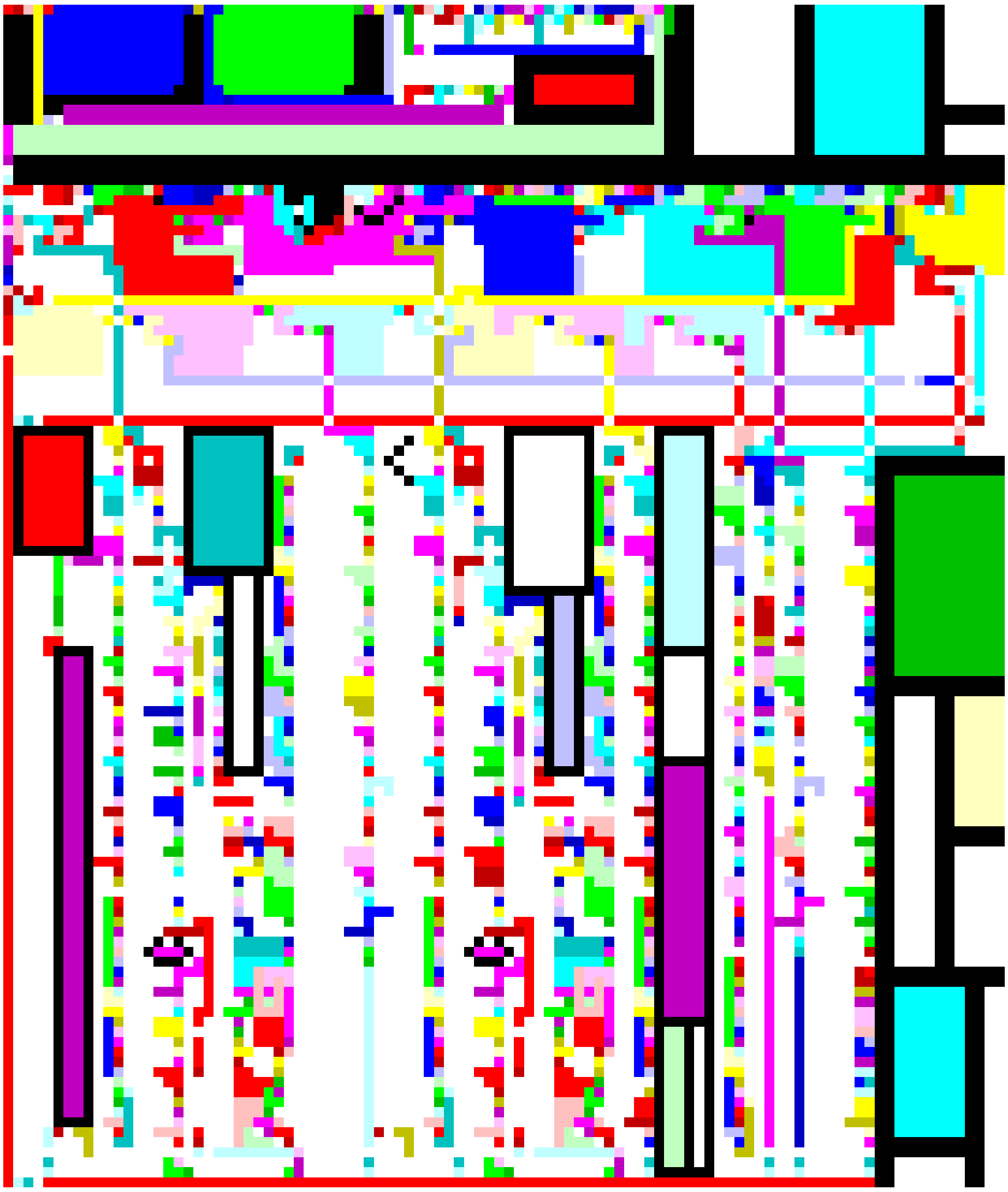

Figure B.15: *piet_bfi_16 - A brainfuck interpreter, by Matthias Ernst (Morgan-Mar 2023a).*

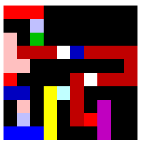

Figure B.16: *euclid_clint - Euclid's algorithm (likely broken), by Clint Herron (Morgan-Mar 2023a).*

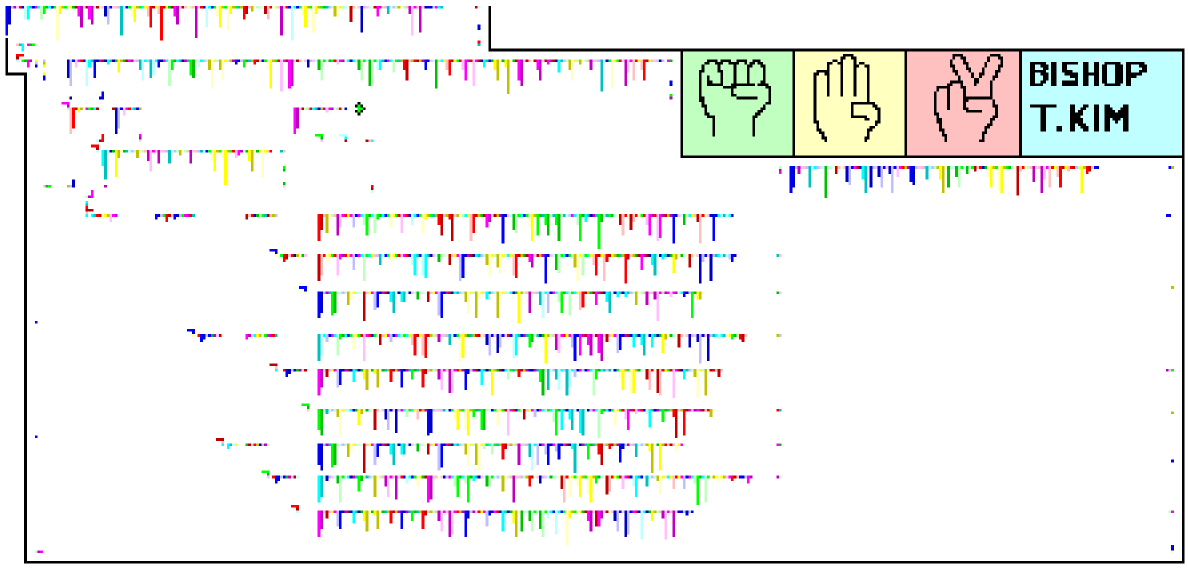

Figure B.17: *rockpaperscissors – A rock paper scissors game, by Patrick Bishop and Tim Kim (Morgan-Mar 2023*a*).*

C Input and Outputs for Evaluation Code

99bottles:

```
99bottles of beer on the wall, 99bottles of beer.
Take one down and pass it around, 98bottles of beer on the wall.
98bottles of beer on the wall, 98bottles of beer.
Take one down and pass it around, 97bottles of beer on the wall.
97bottles of beer on the wall, 97bottles of beer.
Take one down and pass it around, 96bottles of beer on the wall.
96bottles of beer on the wall, 96bottles of beer.
Take one down and pass it around, 95bottles of beer on the wall.
95bottles of beer on the wall, 95bottles of beer.
Take one down and pass it around, 94bottles of beer on the wall.
94bottles of beer on the wall, 94bottles of beer.
Take one down and pass it around, 93bottles of beer on the wall.
93bottles of beer on the wall, 93bottles of beer.
Take one down and pass it around, 92bottles of beer on the wall.
92bottles of beer on the wall, 92bottles of beer.
Take one down and pass it around, 91bottles of beer on the wall.
91bottles of beer on the wall, 91bottles of beer.
Take one down and pass it around, 90bottles of beer on the wall.
90bottles of beer on the wall, 90bottles of beer.
Take one down and pass it around, 89bottles of beer on the wall.
89bottles of beer on the wall, 89bottles of beer.
Take one down and pass it around, 88bottles of beer on the wall.
88bottles of beer on the wall, 88bottles of beer.
Take one down and pass it around, 87bottles of beer on the wall.
87bottles of beer on the wall, 87bottles of beer.
Take one down and pass it around, 86bottles of beer on the wall.
86bottles of beer on the wall, 86bottles of beer.
Take one down and pass it around, 85bottles of beer on the wall.
85bottles of beer on the wall, 85bottles of beer.
Take one down and pass it around, 84bottles of beer on the wall.
84bottles of beer on the wall, 84bottles of beer.
Take one down and pass it around, 83bottles of beer on the wall.
83bottles of beer on the wall, 83bottles of beer.
Take one down and pass it around, 82bottles of beer on the wall.
82bottles of beer on the wall, 82bottles of beer.
Take one down and pass it around, 81bottles of beer on the wall.
81bottles of beer on the wall, 81bottles of beer.
Take one down and pass it around, 80bottles of beer on the wall.
80bottles of beer on the wall, 80bottles of beer.
Take one down and pass it around, 79bottles of beer on the wall.
79bottles of beer on the wall, 79bottles of beer.
Take one down and pass it around, 78bottles of beer on the wall.
78bottles of beer on the wall, 78bottles of beer.
Take one down and pass it around, 77bottles of beer on the wall.
77bottles of beer on the wall, 77bottles of beer.
```

```
Take one down and pass it around, 76bottles of beer on the wall.
76bottles of beer on the wall, 76bottles of beer.
Take one down and pass it around, 75bottles of beer on the wall.
75bottles of beer on the wall, 75bottles of beer.
Take one down and pass it around, 74bottles of beer on the wall.
74bottles of beer on the wall, 74bottles of beer.
Take one down and pass it around, 73bottles of beer on the wall.
73bottles of beer on the wall, 73bottles of beer.
Take one down and pass it around, 72bottles of beer on the wall.
72bottles of beer on the wall, 72bottles of beer.
Take one down and pass it around, 71bottles of beer on the wall.
71bottles of beer on the wall, 71bottles of beer.
Take one down and pass it around, 70bottles of beer on the wall.
70bottles of beer on the wall, 70bottles of beer.
Take one down and pass it around, 69bottles of beer on the wall.
69bottles of beer on the wall, 69bottles of beer.
Take one down and pass it around, 68bottles of beer on the wall.
68bottles of beer on the wall, 68bottles of beer.
Take one down and pass it around, 67bottles of beer on the wall.
67bottles of beer on the wall, 67bottles of beer.
Take one down and pass it around, 66bottles of beer on the wall.
66bottles of beer on the wall, 66bottles of beer.
Take one down and pass it around, 65bottles of beer on the wall.
65bottles of beer on the wall, 65bottles of beer.
Take one down and pass it around, 64bottles of beer on the wall.
64bottles of beer on the wall, 64bottles of beer.
Take one down and pass it around, 63bottles of beer on the wall.
63bottles of beer on the wall, 63bottles of beer.
Take one down and pass it around, 62bottles of beer on the wall.
62bottles of beer on the wall, 62bottles of beer.
Take one down and pass it around, 61bottles of beer on the wall.
61bottles of beer on the wall, 61bottles of beer.
Take one down and pass it around, 60bottles of beer on the wall.
60bottles of beer on the wall, 60bottles of beer.
Take one down and pass it around, 59bottles of beer on the wall.
59bottles of beer on the wall, 59bottles of beer.
Take one down and pass it around, 58bottles of beer on the wall.
58bottles of beer on the wall, 58bottles of beer.
Take one down and pass it around, 57bottles of beer on the wall.
57bottles of beer on the wall, 57bottles of beer.
Take one down and pass it around, 56bottles of beer on the wall.
56bottles of beer on the wall, 56bottles of beer.
Take one down and pass it around, 55bottles of beer on the wall.
55bottles of beer on the wall, 55bottles of beer.
Take one down and pass it around, 54bottles of beer on the wall.
54bottles of beer on the wall, 54bottles of beer.
Take one down and pass it around, 53bottles of beer on the wall.
53bottles of beer on the wall, 53bottles of beer.
Take one down and pass it around, 52bottles of beer on the wall.
52bottles of beer on the wall, 52bottles of beer.
Take one down and pass it around, 51bottles of beer on the wall.
51bottles of beer on the wall, 51bottles of beer.
Take one down and pass it around, 50bottles of beer on the wall.
50bottles of beer on the wall, 50bottles of beer.
Take one down and pass it around, 49bottles of beer on the wall.
49bottles of beer on the wall, 49bottles of beer.
Take one down and pass it around, 48bottles of beer on the wall.
```

```
48bottles of beer on the wall, 48bottles of beer.
Take one down and pass it around, 47bottles of beer on the wall.
47bottles of beer on the wall, 47bottles of beer.
Take one down and pass it around, 46bottles of beer on the wall.
46bottles of beer on the wall, 46bottles of beer.
Take one down and pass it around, 45bottles of beer on the wall.
45bottles of beer on the wall, 45bottles of beer.
Take one down and pass it around, 44bottles of beer on the wall.
44bottles of beer on the wall, 44bottles of beer.
Take one down and pass it around, 43bottles of beer on the wall.
43bottles of beer on the wall, 43bottles of beer.
Take one down and pass it around, 42bottles of beer on the wall.
42bottles of beer on the wall, 42bottles of beer.
Take one down and pass it around, 41bottles of beer on the wall.
41bottles of beer on the wall, 41bottles of beer.
Take one down and pass it around, 40bottles of beer on the wall.
40bottles of beer on the wall, 40bottles of beer.
Take one down and pass it around, 39bottles of beer on the wall.
39bottles of beer on the wall, 39bottles of beer.
Take one down and pass it around, 38bottles of beer on the wall.
38bottles of beer on the wall, 38bottles of beer.
Take one down and pass it around, 37bottles of beer on the wall.
37bottles of beer on the wall, 37bottles of beer.
Take one down and pass it around, 36bottles of beer on the wall.
36bottles of beer on the wall, 36bottles of beer.
Take one down and pass it around, 35bottles of beer on the wall.
35bottles of beer on the wall, 35bottles of beer.
Take one down and pass it around, 34bottles of beer on the wall.
34bottles of beer on the wall, 34bottles of beer.
Take one down and pass it around, 33bottles of beer on the wall.
33bottles of beer on the wall, 33bottles of beer.
Take one down and pass it around, 32bottles of beer on the wall.
32bottles of beer on the wall, 32bottles of beer.
Take one down and pass it around, 31bottles of beer on the wall.
31bottles of beer on the wall, 31bottles of beer.
Take one down and pass it around, 30bottles of beer on the wall.
30bottles of beer on the wall, 30bottles of beer.
Take one down and pass it around, 29bottles of beer on the wall.
29bottles of beer on the wall, 29bottles of beer.
Take one down and pass it around, 28bottles of beer on the wall.
28bottles of beer on the wall, 28bottles of beer.
Take one down and pass it around, 27bottles of beer on the wall.
27bottles of beer on the wall, 27bottles of beer.
Take one down and pass it around, 26bottles of beer on the wall.
26bottles of beer on the wall, 26bottles of beer.
Take one down and pass it around, 25bottles of beer on the wall.
25bottles of beer on the wall, 25bottles of beer.
Take one down and pass it around, 24bottles of beer on the wall.
24bottles of beer on the wall, 24bottles of beer.
Take one down and pass it around, 23bottles of beer on the wall.
23bottles of beer on the wall, 23bottles of beer.
Take one down and pass it around, 22bottles of beer on the wall.
22bottles of beer on the wall, 22bottles of beer.
Take one down and pass it around, 21bottles of beer on the wall.
21bottles of beer on the wall, 21bottles of beer.
Take one down and pass it around, 20bottles of beer on the wall.
20bottles of beer on the wall, 20bottles of beer.
```

```
Take one down and pass it around, 19bottles of beer on the wall.
19bottles of beer on the wall, 19bottles of beer.
Take one down and pass it around, 18bottles of beer on the wall.
18bottles of beer on the wall, 18bottles of beer.
Take one down and pass it around, 17bottles of beer on the wall.
17bottles of beer on the wall, 17bottles of beer.
Take one down and pass it around, 16bottles of beer on the wall.
16bottles of beer on the wall, 16bottles of beer.
Take one down and pass it around, 15bottles of beer on the wall.
15bottles of beer on the wall, 15bottles of beer.
Take one down and pass it around, 14bottles of beer on the wall.
14bottles of beer on the wall, 14bottles of beer.
Take one down and pass it around, 13bottles of beer on the wall.
13bottles of beer on the wall, 13bottles of beer.
Take one down and pass it around, 12bottles of beer on the wall.
12bottles of beer on the wall, 12bottles of beer.
Take one down and pass it around, 11bottles of beer on the wall.
11bottles of beer on the wall, 11bottles of beer.
Take one down and pass it around, 10bottles of beer on the wall.
10bottles of beer on the wall, 10bottles of beer.
Take one down and pass it around, 9bottles of beer on the wall.
9bottles of beer on the wall, 9bottles of beer.
Take one down and pass it around, 8bottles of beer on the wall.
8bottles of beer on the wall, 8bottles of beer.
Take one down and pass it around, 7bottles of beer on the wall.
7bottles of beer on the wall, 7bottles of beer.
Take one down and pass it around, 6bottles of beer on the wall.
6bottles of beer on the wall, 6bottles of beer.
Take one down and pass it around, 5bottles of beer on the wall.
5bottles of beer on the wall, 5bottles of beer.
Take one down and pass it around, 4bottles of beer on the wall.
4bottles of beer on the wall, 4bottles of beer.
Take one down and pass it around, 3bottles of beer on the wall.
3bottles of beer on the wall, 3bottles of beer.
Take one down and pass it around, 2bottles of beer on the wall.
2bottles of beer on the wall, 2bottles of beer.
Take one down and pass it around, 1bottle of beer on the wall.
1bottle of beer on the wall, 1bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99bottles of beer on the wall.
```

Result provided by: npiet, Zago's interpreter, rpiet,Magpie

```
99
bottles of beer on the wall, 99
bottles of beer.
Take one down and pass it around, 98
bottles of beer on the wall.
98
bottles of beer on the wall, 98
bottles of beer.
Take one down and pass it around, 97
bottles of beer on the wall.
97
```

```
bottles of beer on the wall, 97
bottles of beer.
Take one down and pass it around, 96
bottles of beer on the wall.
96
bottles of beer on the wall, 96
bottles of beer.
Take one down and pass it around, 95
bottles of beer on the wall.
95
bottles of beer on the wall, 95
bottles of beer.
Take one down and pass it around, 94
bottles of beer on the wall.
94
bottles of beer on the wall, 94
bottles of beer.
Take one down and pass it around, 93
bottles of beer on the wall.
93
bottles of beer on the wall, 93
bottles of beer.
Take one down and pass it around, 92
bottles of beer on the wall.
92
bottles of beer on the wall, 92
bottles of beer.
Take one down and pass it around, 91
bottles of beer on the wall.
91
bottles of beer on the wall, 91
bottles of beer.
Take one down and pass it around, 90
bottles of beer on the wall.
90
bottles of beer on the wall, 90
bottles of beer.
Take one down and pass it around, 89
bottles of beer on the wall.
89
bottles of beer on the wall, 89
bottles of beer.
Take one down and pass it around, 88
bottles of beer on the wall.
88
bottles of beer on the wall, 88
bottles of beer.
Take one down and pass it around, 87
bottles of beer on the wall.
87
bottles of beer on the wall, 87
bottles of beer.
Take one down and pass it around, 86
bottles of beer on the wall.
86
bottles of beer on the wall, 86
bottles of beer.
```

```
Take one down and pass it around, 85
bottles of beer on the wall.
85
bottles of beer on the wall, 85
bottles of beer.
Take one down and pass it around, 84
bottles of beer on the wall.
84
bottles of beer on the wall, 84
bottles of beer.
Take one down and pass it around, 83
bottles of beer on the wall.
83
bottles of beer on the wall, 83
bottles of beer.
Take one down and pass it around, 82
bottles of beer on the wall.
82
bottles of beer on the wall, 82
bottles of beer.
Take one down and pass it around, 81
bottles of beer on the wall.
81
bottles of beer on the wall, 81
bottles of beer.
Take one down and pass it around, 80
bottles of beer on the wall.
80
bottles of beer on the wall, 80
bottles of beer.
Take one down and pass it around, 79
bottles of beer on the wall.
79
bottles of beer on the wall, 79
bottles of beer.
Take one down and pass it around, 78
bottles of beer on the wall.
78
bottles of beer on the wall, 78
bottles of beer.
Take one down and pass it around, 77
bottles of beer on the wall.
77
bottles of beer on the wall, 77
bottles of beer.
Take one down and pass it around, 76
bottles of beer on the wall.
76
bottles of beer on the wall, 76
bottles of beer.
Take one down and pass it around, 75
bottles of beer on the wall.
75
bottles of beer on the wall, 75
bottles of beer.
Take one down and pass it around, 74
bottles of beer on the wall.
```

```
74
bottles of beer on the wall, 74
bottles of beer.
Take one down and pass it around, 73
bottles of beer on the wall.
73
bottles of beer on the wall, 73
bottles of beer.
Take one down and pass it around, 72
bottles of beer on the wall.
72
bottles of beer on the wall, 72
bottles of beer.
Take one down and pass it around, 71
bottles of beer on the wall.
71
bottles of beer on the wall, 71
bottles of beer.
Take one down and pass it around, 70
bottles of beer on the wall.
70
bottles of beer on the wall, 70
bottles of beer.
Take one down and pass it around, 69
bottles of beer on the wall.
69
bottles of beer on the wall, 69
bottles of beer.
Take one down and pass it around, 68
bottles of beer on the wall.
68
bottles of beer on the wall, 68
bottles of beer.
Take one down and pass it around, 67
bottles of beer on the wall.
67
bottles of beer on the wall, 67
bottles of beer.
Take one down and pass it around, 66
bottles of beer on the wall.
66
bottles of beer on the wall, 66
bottles of beer.
Take one down and pass it around, 65
bottles of beer on the wall.
65
bottles of beer on the wall, 65
bottles of beer.
Take one down and pass it around, 64
bottles of beer on the wall.
64
bottles of beer on the wall, 64
bottles of beer.
Take one down and pass it around, 63
bottles of beer on the wall.
63
bottles of beer on the wall, 63
bottles of beer.
Take one down and pass it around, 62
bottles of beer on the wall.
62
bottles of beer on the wall, 62
bottles of beer.
Take one down and pass it around, 61
bottles of beer on the wall.
61
bottles of beer on the wall, 61
bottles of beer.
Take one down and pass it around, 60
bottles of beer on the wall.
60
bottles of beer on the wall, 60
bottles of beer.
Take one down and pass it around, 59
bottles of beer on the wall.
59
bottles of beer on the wall, 59
bottles of beer.
Take one down and pass it around, 58
bottles of beer on the wall.
58
bottles of beer on the wall, 58
bottles of beer.
Take one down and pass it around, 57
bottles of beer on the wall.
57
bottles of beer on the wall, 57
bottles of beer.
Take one down and pass it around, 56
bottles of beer on the wall.
56
bottles of beer on the wall, 56
bottles of beer.
Take one down and pass it around, 55
bottles of beer on the wall.
55
bottles of beer on the wall, 55
bottles of beer.
Take one down and pass it around, 54
bottles of beer on the wall.
54
bottles of beer on the wall, 54
bottles of beer.
Take one down and pass it around, 53
bottles of beer on the wall.
53
bottles of beer on the wall, 53
bottles of beer.
Take one down and pass it around, 52
bottles of beer on the wall.
52
bottles of beer on the wall, 52
bottles of beer.
Take one down and pass it around, 51
```

```
bottles of beer on the wall.
51
bottles of beer on the wall, 51
bottles of beer.
Take one down and pass it around, 50
bottles of beer on the wall.
50
bottles of beer on the wall, 50
bottles of beer.
Take one down and pass it around, 49
bottles of beer on the wall.
49
bottles of beer on the wall, 49
bottles of beer.
Take one down and pass it around, 48
bottles of beer on the wall.
48
bottles of beer on the wall, 48
bottles of beer.
Take one down and pass it around, 47
bottles of beer on the wall.
47
bottles of beer on the wall, 47
bottles of beer.
Take one down and pass it around, 46
bottles of beer on the wall.
46
bottles of beer on the wall, 46
bottles of beer.
Take one down and pass it around, 45
bottles of beer on the wall.
45
bottles of beer on the wall, 45
bottles of beer.
Take one down and pass it around, 44
bottles of beer on the wall.
44
bottles of beer on the wall, 44
bottles of beer.
Take one down and pass it around, 43
bottles of beer on the wall.
43
bottles of beer on the wall, 43
bottles of beer.
Take one down and pass it around, 42
bottles of beer on the wall.
42
bottles of beer on the wall, 42
bottles of beer.
Take one down and pass it around, 41
bottles of beer on the wall.
41
bottles of beer on the wall, 41
bottles of beer.
Take one down and pass it around, 40
bottles of beer on the wall.
40
```

```
bottles of beer on the wall, 40
bottles of beer.
Take one down and pass it around, 39
bottles of beer on the wall.
39
bottles of beer on the wall, 39
bottles of beer.
Take one down and pass it around, 38
bottles of beer on the wall.
38
bottles of beer on the wall, 38
bottles of beer.
Take one down and pass it around, 37
bottles of beer on the wall.
37
bottles of beer on the wall, 37
bottles of beer.
Take one down and pass it around, 36
bottles of beer on the wall.
36
bottles of beer on the wall, 36
bottles of beer.
Take one down and pass it around, 35
bottles of beer on the wall.
35
bottles of beer on the wall, 35
bottles of beer.
Take one down and pass it around, 34
bottles of beer on the wall.
34
bottles of beer on the wall, 34
bottles of beer.
Take one down and pass it around, 33
bottles of beer on the wall.
33
bottles of beer on the wall, 33
bottles of beer.
Take one down and pass it around, 32
bottles of beer on the wall.
32
bottles of beer on the wall, 32
bottles of beer.
Take one down and pass it around, 31
bottles of beer on the wall.
31
bottles of beer on the wall, 31
bottles of beer.
Take one down and pass it around, 30
bottles of beer on the wall.
30
bottles of beer on the wall, 30
bottles of beer.
Take one down and pass it around, 29
bottles of beer on the wall.
29
bottles of beer on the wall, 29
bottles of beer.
```

```
Take one down and pass it around, 28
bottles of beer on the wall.
28
bottles of beer on the wall, 28
bottles of beer.
Take one down and pass it around, 27
bottles of beer on the wall.
27
bottles of beer on the wall, 27
bottles of beer.
Take one down and pass it around, 26
bottles of beer on the wall.
26
bottles of beer on the wall, 26
bottles of beer.
Take one down and pass it around, 25
bottles of beer on the wall.
25
bottles of beer on the wall, 25
bottles of beer.
Take one down and pass it around, 24
bottles of beer on the wall.
24
bottles of beer on the wall, 24
bottles of beer.
Take one down and pass it around, 23
bottles of beer on the wall.
23
bottles of beer on the wall, 23
bottles of beer.
Take one down and pass it around, 22
bottles of beer on the wall.
22
bottles of beer on the wall, 22
bottles of beer.
Take one down and pass it around, 21
bottles of beer on the wall.
21
bottles of beer on the wall, 21
bottles of beer.
Take one down and pass it around, 20
bottles of beer on the wall.
20
bottles of beer on the wall, 20
bottles of beer.
Take one down and pass it around, 19
bottles of beer on the wall.
19
bottles of beer on the wall, 19
bottles of beer.
Take one down and pass it around, 18
bottles of beer on the wall.
18
bottles of beer on the wall, 18
bottles of beer.
Take one down and pass it around, 17
bottles of beer on the wall.
```

```
17
bottles of beer on the wall, 17
bottles of beer.
Take one down and pass it around, 16
bottles of beer on the wall.
16
bottles of beer on the wall, 16
bottles of beer.
Take one down and pass it around, 15
bottles of beer on the wall.
15
bottles of beer on the wall, 15
bottles of beer.
Take one down and pass it around, 14
bottles of beer on the wall.
14
bottles of beer on the wall, 14
bottles of beer.
Take one down and pass it around, 13
bottles of beer on the wall.
13
bottles of beer on the wall, 13
bottles of beer.
Take one down and pass it around, 12
bottles of beer on the wall.
12
bottles of beer on the wall, 12
bottles of beer.
Take one down and pass it around, 11
bottles of beer on the wall.
11
bottles of beer on the wall, 11
bottles of beer.
Take one down and pass it around, 10
bottles of beer on the wall.
10
bottles of beer on the wall, 10
bottles of beer.
Take one down and pass it around, 9
bottles of beer on the wall.
9
bottles of beer on the wall, 9
bottles of beer.
Take one down and pass it around, 8
bottles of beer on the wall.
8
bottles of beer on the wall, 8
bottles of beer.
Take one down and pass it around, 7
bottles of beer on the wall.
7
bottles of beer on the wall, 7
bottles of beer.
Take one down and pass it around, 6
bottles of beer on the wall.
6
bottles of beer on the wall, 6
bottles of beer.
Take one down and pass it around, 5
bottles of beer on the wall.
5
bottles of beer on the wall, 5
bottles of beer.
Take one down and pass it around, 4
bottles of beer on the wall.
4
bottles of beer on the wall, 4
bottles of beer.
Take one down and pass it around, 3
bottles of beer on the wall.
3
bottles of beer on the wall, 3
bottles of beer.
Take one down and pass it around, 2
bottles of beer on the wall.
2
bottles of beer on the wall, 2
bottles of beer.
Take one down and pass it around, 1
bottle of beer on the wall.
1
bottle of beer on the wall, 1
bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.
No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99
bottles of beer on the wall.
```

Result provided by: Ynn's interpreter. Bouman's interpreter gave no result.
Countdown:

```
10
9
8
7
6
5
4
3
2
1
```

Result produced by: npiet, Zago's interpreter, Bouman's interpreter,rpiet, Magpie

```
10

9

8
```

```
7

6

5

4

3

2

1
```

Result produced by: Ynn’s interpreter

Cowsay:
Input: This is a cow speaking!

```

_________________________
is is a cow speaking! >
---------------------
   \ ^__^
   \ (oo)\_______
      (__)\ )\/\
         ||----w |
         || ||
```

Result produced by: npiet, Zago’s interpreter, rpiet, Magpie. Bouman’s interpreter panicked, Ynn’s interpreter timed out.
Input: Język programowania – zbiór zasad określających[....]

```

_________________________________________
zyk programowania  zbir zasad o \
elajcych[....] /
-------------------------------------
   \ ^__^
   \ (oo)\_______
      (__)\ )\/\
         ||----w |
         || ||
```

Result produced by: npiet, Zago’s interpreter, Magpie

```

_________________________________________
zyk programowania  zbir zasad o \
eaj
[....] /
-------------------------------------
   \ ^__^
   \ (oo)\_______
      (__)\ )\/\
         ||----w |
         || ||
```

Result produced by: rpiet. Ynn’s interpreter timed out and Bouman’s interpreter panicked.

Fizzbuzz

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
```

Result produced by: npiet, Zago’s interpreter, rpiet, Magpie

```
1

2

Fizz
4

Buzz
Fizz
7

8

Fizz
Buzz
11

Fizz
13

14

FizzBuzz
16
```

Result produced by: Ynn's interpreter. Bouman's interpreter returned no output.
GameOfLife:
Input: 1\n2\n2\n1\n2\n3\n1\n3\n3\n1\n1\n2\n0\n0\n0\n2\n

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
0)STEP
1)TOGGLE
2)QUITXY
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . o .
. . . . . . . . . .
0)STEP
1)TOGGLE
2)QUITXY
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . o .
. . . . . . . . o .
. . . . . . . . . .
0)STEP
1)TOGGLE
2)QUITXY
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . o o .
. . . . . . . . o .
. . . . . . . . . .
0)STEP
1)TOGGLE
2)QUITXY
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . o o .
. . . . . . . . o o
. . . . . . . . . .
0)STEP
1)TOGGLE
2)QUIT
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . o o o
. . . . . . . o o o
. . . . . . . . . .
0)STEP
1)TOGGLE
2)QUIT
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . o .
. . . . . . . o . o
. . . . . . . o . o
. . . . . . . . o .
0)STEP
1)TOGGLE
2)QUIT
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . o .
. . . . . . . o . o
. . . . . . . o . o
. . . . . . . . o .
0)STEP
1)TOGGLE
2)QUIT
```

Result produced by: npiet, Zago's interpreter, Ynn's interpreter, Magpie. Bouman's interpreter panicked, and rpiet timed out.

pietquest
Input: 2\n1\n1\n2\n2\n2\n1\n2\n1\n1\n1\n1\n2\n3\n1\n1\n1\n

```
================
= Piet’s Quest =
================

You find yourself in a rather dark studio.
There is an easel here.
There is a ladder leading down.

Please select:

1 - paint
2 - go down ladder

You find yourself in a dusty, dim hallway.
There is a door to the kitchen here.
There is a door to the bedroom here.
There is a rickety loft ladder here.

Where do you want to go today?
1 - kitchen
2 - bedroom
3 - loft

You find yourself in a well-stocked kitchen.
It smells invitingly of apple pancake.
Your wife is here.
She gives you a look.

Your options:
1 - talk to her
2 - go back to the hallway

You find yourself in conversation with your wife.
You: Dear, may I borrow that pastry brush?
Wife: Not again! You know how hard it is to get the paint out.
You: Pleeease?
Wife: ...oh, very well. I hope you realise this means no pie for you tonight.

You are now the proud owner of a pastry brush.

You find yourself in a well-stocked kitchen.
It smells invitingly of apple pancake.
Your wife is here.
She gives you a look.

Your options:
1 - talk to her
2 - go back to the hallway

You find yourself in a dusty, dim hallway.
There is a door to the kitchen here.
There is a door to the bedroom here.
There is a rickety loft ladder here.

Where do you want to go today?
1 - kitchen
```

```
2 - bedroom
3 - loft

You find yourself in a cozy bedroom. Sunbeams warm the carefully made bed.

Your options:
1 - return to the hallway
2 - rip off the bedclothes

You find yourself in growing excitement.
You look around sneakily; your eyes begin to glow. You pounce.
Pounce!
Feathers fly everywhere.
You snatch the brilliant white bedsheet and hold it close to your chest.
You have a white sheet.

You find yourself in a cozy bedroom. The harsh sunlight exposes the unmade bed.
You feel a little guilty.

Your options:
1 - return to the hallway

You find yourself in a dusty, dim hallway.
There is a door to the kitchen here.
There is a door to the bedroom here.
There is a rickety loft ladder here.

Where do you want to go today?
1 - kitchen
2 - bedroom
3 - loft

You find yourself in a cozy bedroom. The harsh sunlight exposes the unmade bed.
You feel a little guilty.

Your options:
1 - return to the hallway

You find yourself in a dusty, dim hallway.
There is a door to the kitchen here.
There is a door to the bedroom here.
There is a rickety loft ladder here.

Where do you want to go today?
1 - kitchen
2 - bedroom
3 - loft

You find yourself in a well-stocked kitchen.
It smells invitingly of apple pancake.
Your wife is here.
She gives you a look.

Your options:
1 - talk to her
2 - go back to the hallway
```

```
You find yourself in conversation with your wife.
You: Dear, what should I paint next?
Wife: Oh, you are a silly.
Wife: Just draw some lines or something.
Wife: So long as you sign it, they’ll pay you.
You: Lines! There’s an idea.

You find yourself in a quandary.
Type 1 to continue.

Wife: You still look puzzled.
You: ...what should I call it?
Wife: Why don’t you paint it first, and show me? I’m sure I’ll come up with
    something to solve your puzzlement.
You: ...you are so sensible.

You are filled with new purpose!

You find yourself in a well-stocked kitchen.
It smells invitingly of apple pancake.
Your wife is here.
She gives you a look.

Your options:
1 - talk to her
2 - go back to the hallway

You find yourself in a dusty, dim hallway.
There is a door to the kitchen here.
There is a door to the bedroom here.
There is a rickety loft ladder here.

Where do you want to go today?
1 - kitchen
2 - bedroom
3 - loft

You find yourself in a rather dark studio.
There is an easel here.
There is a ladder leading down.

Please select:

1 - paint
2 - go down ladder

You find yourself in a trance...

By and by, you come to.
Everything is covered in paint.
You look up.

You behold your magnificent creation!

It lacks only a name.

1 -call your wife
```

```
Your long-suffering wife climbs the ladder.
She winces as she sees the loft.
Gleefully, you point at your masterpiece.
She consideres it in silence.

You: The story of today, of how this work came to be, of all you said to me - it
    is in these lines.
You: To one who knows how to read it, everything will be plain.
Wife: You mean it’s some sort of code?
You: It’s art!

Your wife purses her lips and looks around again.

1 - continue
Wife: A mess, that’s what I call it. Where’s my pastry brush?
You: Right here!

You dip the brush and scrawl across the top: MESS

Your quest is complete.
```

Result produced by: npiet, Zago’s interpreter, Magpie. Bouman’s interpreter panicked and Ynn’s interpreter timed out.

rockpaperscissors
Input: P\nR\nS\nQ

```
Let’s Play Rock Paper Scissors!
Input ’P’, ’S’, or ’R’ to play or ’Q’ to quit: Scissors beats Paper: CPU wins!
Score: CPU: 1 You: 0
Input ’P’, ’S’, or ’R’ to play or ’Q’ to quit: CPU chose Rock: It’s a tie!
Input ’P’, ’S’, or ’R’ to play or ’Q’ to quit: Rock beats Scissors: CPU wins!
Score: CPU: 2 You: 0
Input ’P’, ’S’, or ’R’ to play or ’Q’ to quit:
```

Result produced by: npiet, Zago’s interpreter, rpiet, Magpie. Ynn’s interpreter & Bouman’s interpreter panicked.

D Filesystem

The project file system is laid out as follows:

- `Magpie` – Magpie folder
- `tableau.py` – The evaluation python script
- `imgs` – A small collection of test images.
- `README.md` – the readme file.

Magpie is laid out as follows:

- `.ocamlformat`
- `dune`
- `magpie.opam` – opam files
- `dune-project` – helper files for dune
- `bin` – binary folder
  - `dune` – stores bin folder information
  - `main.ml` – entry file to the library.
- `lib` – library folder
  - `dune` – stores Magpie library information
  - `colour.ml` – Stores the colours Piet supports and the Delta function
  - `Colour.mli` – interface file for colour.ml
  - `flowfunctions.ml` – Movement functions, including the flood-fill algorithm.
  - `flowfunctions.mli` – interface file for flowfunctions.ml
  - `Magpie_init.ml` – Code for idle and eval states, argument parser, find_next
  - `Magpie_init.mli` – interface file for Magpie_init.ml
  - `magpie.ml` – holds renamings for the modules
  - `pietops.ml` – Specification operations, and calcop function.
  - `pietops.mli` – interface file for pietops.ml
  - `read.ml` – Reads a file and converts it to Piet's colour system
  - `read.mli` – interface file for read.ml
  - `state.ml` – Definition of the state vector.
  - `state.mli` – interface file for state.ml

`img` contains the evaluation images used in this paper.