The Art Of Computer Programming: A formal specification for Piet

Nimrod Libman

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2022

Abstract

We discuss formal specifications of programming languages, and small-step operational semantics as a method for formal specification. We then discuss the $\mathbb{K}$ framework, an approach to defining formal semantics. We then detail the esoteric programming language Piet, and other related languages called ”fungeoids”. Finally, we build an executable formal semantics for Piet using the $\mathbb{K}$ framework, and consider how the unusual behaviours of this language relate to how it is specified.

Acknowledgements

I would like to thank my supervisor, Claudia, for proposing this project and introducing me to operational semantics, and for pointing me in the right directions whenever I was lost.

Table of Contents

Chapter 1

Introduction

1.1 Formal Semantics

When specifying a programming language, language designers must define the *syntax* and *semantics* of the language. Specifying syntax is a "solved" problem; defining a Context-Free Grammar is sufficient, as these are unambiguous and studied well enough such that creating a parser for this grammar is easy to implement without bugs. Because of this, most specification documents devote a lot of space on describing the behaviours of the language. However, these are usually done in natural language, which is known to be ambiguous. This can lead to undefined behaviours in edge-cases, or even in multiple implementations having different behaviours for what is defined in the specification; both of which are correct by *some* reading of the text.

Therefore, it is of value to define the semantics of a language formally. By doing so, we eliminate divergences of behaviour between implementations. In addition, by defining behaviours in terms of logic, places where behaviour is not defined could be more easily spotted, so that the language designer can come to a decision as to the intended behaviour in these cases.

1.2 The $\mathbb{K}$ Framework

The $\mathbb{K}$ framework [18][6] is a project that seeks to provide a language for detailing the formal semantics of any programming language. It uses rewrite rules to define the behaviour of the specified language; if the current program state matches parameters defined in the rule - for example, the current instruction, or a variable is of a given value - then the rule is applied, and the state is changed as detailed by the rule. By modelling execution of programs in this way, when one writes a specification using the $\mathbb{K}$ framework, the specification can be turned into an interpreter for the language.

1.3 Piet

Piet[14] is an esoteric language designed by David Morgan-Mar, and is best known for using bitmap images as source code. This unique quality makes Piet amongst some of the better-known esoteric programming languages. Piet's use of images as code makes it particularly interesting as this makes the instruction space of the language "two-dimensional"; instructions must be addressed by their position as an (x, y) coordinate rather than an index in a list of instructions, like most conventional languages. This quality places Piet in the category of languages called *fungeoids*, defined by this trait of two-dimensional instruction space.

This trait also brings with it another quality; the idea of a "direction" of execution. Whereas in a traditional language programs increment the program counter to access the next instruction (branch and loop constructs notwithstanding), a fungeoid must keep track of a direction pointer, representing if the program is flowing along the X or Y axes, and whether in the positive or negative direction. Flow control is handled with this construct in Piet; the direction pointer is adjusted so that the instruction pointer arrives back at the same instruction to implement a loop, for example.

1.4 The value of fungeoid semantics

Fungeoids, like Piet, are all esoteric languages, and as such their primary use is in light-hearted programming competitions rather than real-world use. To provide a formal semantics for such languages would comprise a sort of "stress test" for the $\mathbb{K}$ framework, due to the unusual approach to instruction addressing. A successful implementation of Piet would suggest that the paradigm in which $\mathbb{K}$ operates is general enough to capture the behaviours of these languages, and is therefore suitable to intuitively describing the behaviours of other unusual (and hopefully more useful) languages.

Chapter 2

Previous solutions

Piet is a reasonably well-known esoteric language, and as such there are a lot of existing implementations. I analysed these solutions to get a better understanding of the intentions of the specification, and of ways of implementing certain features.

The main features I was interested in seeing how these solutions solved were the block-building functions, and how they handled black and white pixels (the function of these features is explained below in Section 5).

2.1 Comparisons of various Piet implementations

The most well-established interpreter is `npiet` [19]. There are many other interpreters in existence, however, `npiet` is a representative example. All the interpreters that I have analysed during my research operate on the same basis as it, with minor differences mostly emerging in the virtual machine backend; for example, the handling of input operations. However, there is a general agreement as to the general behaviours such as pathing and the simpler VM operations, although this was not always the case; according to the ”samples” page on the Piet website, `npiet`'s pathing behaviour used to diverge from that of most other interpreters, resulting in unintended program flow for certain programs. Being the most well-established, I shall mostly refer back to modern `npiet`: the behaviours mentioned will more than likely apply to most Piet interpreters.

While interpreters are more common, there also exist a small number of compilers for Piet. `repiet` [1] is one of these compilers, written in `python`, with compile targets of `C`, `C++`, and `python`. As opposed to the interpreted solutions, which compute the colour blocks upon entering them, `repiet` precomputes each colour block in the program before execution, including blocks that are not entered during a given execution, or cannot be entered at all, for example, blocks surrounded by black pixels.

2.1.1 Program blocks

`npiet` uses a recursive expanding-frontier method that updates a pair of variables marking the most extreme positions in the block. Whenever the function finds a new pixel in the same block, with a position that is further in the direction that is specified by the Direction Pointer, then it updates these variables to the new pixel's position. In the case of a tie, the Codel Chooser, a secondary direction pointer, is used to break ties - it chooses a direction perpendicular to the Direction Pointer, the pixel that is furthest along that axis is considered the more extreme and thus the position in which to move to the next block in the next program execution step. This behaviour occurs in the `check_connected_cell()` function in `npiet`.

As `npiet` reaches a new block, it calls this block building function, even if it had already been visited before, it is not cached. This means there is a lot of extra computation when going in loops. It does however avoid calculating unreachable blocks, or those that are not reached during the specific execution of the program.

On the other hand, `repiet` is a compiler and as such precomputes the boundaries of each colour block beforehand, as mentioned before. Then, the compiler finds all the possible transitions between colour blocks; this gives a node graph. Given that one knows the path program execution will take given a certain direction on entry, barring certain operations that modify the Direction Pointer (and the Codel Chooser; hereafter collectively referred to as the direction pointers because their functions are so closely related), one can split each node into a new node for each state of the direction pointers, from each of which there is only one possible next node. In this way the node graph is untangled into a tree, like a traditional compiler's internal abstract syntax tree representation. This is then rewritten as code in the "traditional" languages, which can handle such program flow by design.

In both solutions outlined above, white pixels, which function as "empty space" that the program travels through (this will be discussed in greater depth in the next chapter), are treated as individual 1-pixel blocks in their own right. This allows the inter-block movement semantics to be used to handle the white pixel movement, but requires exceptions to be made in terms of translating what operations the transitions between blocks result in. Presumably, this is easier than the alternative of writing a new set of movement rules for white pixels that allow for the use of the same instruction backend. This informed my decisions on how to implement the movement rules in my specification.

2.1.2 Virtual machine

Both interpreter and compiler solutions can be seen as using a Virtual machine architecture; this is most obvious in the former, when a transition between two coloured blocks occur, the interpreter executes some operations on the stack. When a transition occurs between two blocks, the interpreter generates a virtual machine instruction, which is then executed by the VM, except that here, unlike most virtual machine architectures like the JVM, program flow is handled by the inter-block pathing system rather than by the virtual machine itself - although some operations can influence pathing.

The compiler can also be seen as using a VM, except that here the virtual machine does not execute the instruction; rather, the VM instructions are treated as an internal representation tree. This IR is then rewritten as instructions in other programming languages; a virtual machine being converted to a real machine, as it were.

2.2 Visual tracing

One other implementation of Piet is an interpreter by Jens Bouman [2]. This interpreter does not operate in a manner all that differently from the implementations like `npiet` discussed above; however, it is notable for including a visual tracer. This allowed me to execute programs step by step and see the program flow superimposed on the program image. This was very useful when I was creating test cases, particularly when creating tests to test the program flow features; I was able to verify the behaviour of my tests by running this tracer. It was also of help when debugging certain test cases; by having my implementation log its movements, I was able to compare this with the trace the visual tracer gave me, and find where they were diverging.

Chapter 3

Structural Operational Semantics

There are various ways to define the operational semantics of a language. The $\mathbb{K}$ framework takes an approach known as *Structural Operational Semantics*[10][16]. In this approach we define the behaviour in terms of the specific behaviours of individual syntax elements; when combined, this large number of (hopefully) simple rules should give rise to greater complexity and the full power of the language. This focus on syntax elements allows us to reason about programs in terms of language syntax and therefore the code written, and not concern ourselves with what the parser may or may not happen to do.

3.1 Transition Systems and SMC Machines

The theory of structural operation semantics states that a language can be seen as a *transition system*[16], defined as the tuple $< \Gamma, \rightarrow >$. Here, Γ is the set of *configurations* in which a program can be in, and $\rightarrow$, a binary relation $\Gamma \times \Gamma$, is the set of all *valid transitions* between configurations. Each valid transition is a tuple (e, e') where $e, e' \in \Gamma$, such that the transition system can move from the configuration e to the configuration e'.

Plotkin provides an example of this with the SMC machine evaluating a simple hypothetical programming language `L`. This construct contains two stacks - S, a temporary stack (the Value Stack) used to store symbols that are being processed, and C, the Control Stack which contains the program, with the top of the stack being the first syntax element on the stack. In addition, the machine stores a mapping M from variable identifier symbol to a value. At every step, we look at the head of the stacks to see what rules to apply, and then push new values to the stack based on the symbols seen. In the example below, when faced with an expression $1+2$ on C, this is rewritten as $(1,2,+)$, the 1 is then pushed to S, then the 2 is, and then finally we match the rule where when we see 2 values on S and $+$ on$C$, which pushes the value obtained by adding the top two values on S onto C. One such step is a *transition*.

The SMC machine is therefore a transition system, that contains one rule for every possible symbol that could be at the head of the Control Stack. However, it requires

Snippet 1 The rules for an SMC machine for a language comprised of addition expressions. The first rule moves literals to the value stack, the second looks up variables in M and pushes that value to the stack. The third rule rearranges addition expressions so that the subexpressions can be deconstructed and pushed onto the stack, and the last takes values from the Value stack and adds them together. This can of course be extrapolated to a language of all simple arithmetic expressions, by copying the latter two rules. Reproduced from Plotkin81, Section 1.5.2

$$< S, M, nC > \Rightarrow < nS, M, C >$$

$$< S, M, vC > \Rightarrow < M(v)S, M, C >$$

$$< S, M, e + e'C > \Rightarrow < S, M, ee' + C >$$

$$< m'mS, M, +C > \Rightarrow < nS, M, C > \text{ where } n = m + m'$$

many rules that have little relation to a description of the behaviour of the language the SMC specifies; in the example above, the pushing the left and right hand expressions of the addition expression to the Value Stack is not relevant to our understanding of the addition operation. Instead, this comprises more of an implementation detail, whereas we are concerned more with the ”bigger picture” of what the set of rules is trying to achieve, as that is more useful to us when specifying a language. In fact, we would rather that the aforementioned ”boilerplate” steps be abstracted away.

3.2 Improving the SMC machine

The SMC machine is an unwieldy construct for specifying programming languages, and as such Plotkin later defines a new notation that does not require us to deal with the Value Stack. Here, our preconditions can include a term representing *some* expression that can be obtained through some set of transitions in the SMC machine. For example, we can now define the rule ”If there exists some new expression that an expression on some side of an operator can be rewritten as, through some sequence of transitions, then rewrite the old expression as this expression”, which cannot be defined in an SMC. However, this rule more closely matches our intuitive reasoning about programs (i.e if there exists some subexpression that can be evaluated to another more concise expression, then rewrite it as the concise expression), and is therefore more useful; when evaluating an algebraic expression, we evaluate subexpressions until they become literals, then add the literals.

This relation between e and e', that e can through some (possibly multiple) SMC rules be rewritten as e' is called a *reduction sequence*. A $\mathbb{K}$ rewrite rule is an example of a reduction sequence, as it allows one expression (or rather, a set of configurations containing the expression) to be reduced to another. In addition, reductions sequences are transitive relations, i.e if $e \rightarrow e'$, and $e' \rightarrow e''$, then $e \rightarrow e''$, the reduction sequence can be comprised of several rule applications, each of which is also a reduction sequence.

A Rewrite Rule, the construct used in $\mathbb{K}$, does not have preconditions of the form that small-step semantics presented above can have, like $e \rightarrow e'$; we cannot specify these as

Snippet 2 An example of the rules for evaluating addition expressions. Here, e is an expression that appears in the overall equation, while e' is some expression that does not appear in the program, that e can be reduced to via applying rules in the SMC machine. m represents a terminal expression i.e a literal. The first rule allows us to evaluate the left hand subexpression until it is a literal (and cannot be reduced further). Once the left is a literal, the second then evaluates the right hand subexpression. Once both sides are literals, the addition can occur, and the entire addition expression is replaced with the resulting literal.

$$\frac{e_0 \to e_0'}{e_0 + e_1 \to e_0' + e_1}$$

$$\frac{e_1 \to e_1'}{m_0 + e_1 \to m_0 + e_1'}$$

$$m_0 + m_1 \to m_2 \text{ where } m_2 \text{ is the result of } m_0 + m_1$$

we do not know what e''s concrete value is, and in fact there could be many values for e' for any e. Instead, rewrite rules are applied to an expression whenever they appear at the head of the program, with no such preconditions (although they *can* have simpler preconditions such as "when term X is not equal to 0"). However, this means there is no way to evaluate long or perhaps nested expressions like the addition example above. The rules involving such preconditions are handled via the *strictness annotation*; when we define a syntax element containing some other syntax element, we can mark the expression as "strict" on that subexpression. This is further detailed in Section 4.3.

Chapter 4

The $\mathbb{K}$ Framework

The $\mathbb{K}$ framework is a rewrite-based executable semantic framework. More simply, this means that it gives us tools to define programming languages formally; specifically, in terms of rewrite rules. These are a form of Structural Operational Semantics as described in Chapter 3, where we apply rules that we have defined as transitions between configurations. To specify these transitions and the format of the configuration, K exposes a language to us to write this specification in. Then, we can compile this specification with the `kompile` tool, which generates an interpreter that can execute the specified language. This interpeter allows us to verify the correctness of the specification, by allowing us to run and thus test the specified language.

4.1 Cells

In a $\mathbb{K}$ specification, we define program state within *cells*, which store values or otherwise collections of values; for example, the program code is put into the `<k>` cell. Or, one can define a maping between a variable name and its value within a cell. Together, all cells comprise the program's state. Each cell is given an identifier, as well as other optional attributes; for example, the "multiplicity" attribute which allows us to define many cells of the same structure alongside each other. A use for this attribute would be for an object-oriented language's representation of instantiated objects; we define a cell which contains several subcells within it, such a cell storing a unique identifier, a cell to store member variables, and a cell to store the class the object is a member of. The supercell would then be given the multiplicity attribute specifying that there can be many copies of it; every time an object is instantiated, a new copy of the supercell is created.

These cells together comprise the *configuration* of the language. This directly corresponds to a configuration in Γ in the $< \Gamma, \rightarrow >$ definition of language semantics. Program execution is a sequence of different states of the configuration, where each rule matches expressions within some cell or cells, and writes new information to cells, in preparation for the next rule.

The organisation of the configuration can be used to enforce certain behaviours when

rules are applied; for example, one could store each method within a specific cell, rather than all code in one single cell. Now that the methods are separated from the main body of the program, we can specify that the rules only apply to the cell containing the main program (i.e the <k> cell), rather than evaluating methods before they are called (i.e while the code is in another cell). Similarly, in the object-oriented language example, we create a cell for each instantiated object, and specify in the variable lookup rule to look in the cell corresponding to the currently executing object to get the variables value, rather than in some other cell that the rule could otherwise match.

4.2 Rewrite Rules

A $\mathbb{K}$ rule corresponds to a transition relation as detailed in Chapter 3. A rule consists of a match and a rewrite. First, we search for a pattern of syntax elements within some cells that we specify; for example, a specific statement in the <k> cell, which holds the program code. Then, we specify a pattern to which to rewrite the matched patterns to; this is considered a single transition. This is analogous to a transition relation where the current program state is a configuration in Γ, the set of all possible configurations, and a rewrite rule is a transition relation; a binary relation in the set $\Gamma \times \Gamma$. Indeed, $\mathbb{K}$ refers to the current program state as a configuration, so this analogy is intentional.

Snippet 3 An example of the rules implementing if-statement for a generic imperative language in $\mathbb{K}$. We match the syntax element of the if statement, as well as the literal value of the condition, and then rewrite the whole if-statement syntax element to the appropriate block element.

```
rule if (true)  S:Block else _ => S
rule if (false) _ else S:Block => S
```

Matched patterns need not be literals; one can match abstract syntax elements, and indeed entire subtrees of the abstract syntax tree. This is demonstrated in the example code above; we do not match any literal element within the blocks, but rather, the block element itself. This is useful for defining higher-order operations. This means that a rewrite rule does not directly map to a transition relation as Plotkin describes, but rather to the *set* of transition relations that transition from all possible configurations containing the matched elements.

In the SMC machine, rules can have a reduction relation to some unknown but bound state $e \to e'$. However, calculating e' can often be computationally unfeasible and as such is not supported by $\mathbb{K}$. Instead, rewrite rules can only bind to values that exist in the configuration, and as such preconditions can only check the current configuration rather than bind to some hypothetical future state. However, this means we lose the power that the reduction relation preconditions give us; for example, we lose the ability to evaluate subexpressions as in Snippet 2. For this purpose, $\mathbb{K}$ uses introduce *syntax annotations*, specifically to denote *strictness*, to mimic these behaviours in contexts where rules matching the reduction sequence would be useful. This is detailed further in Section 4.3.

The generated interpreter works by loading the program code (that is, of the program being executed) into the <k> cell, and then repeatedly applying the defined rules until we reach a state where no rule can be applied. At each execution step, the runtime finds some rule that can match some term of set of terms in the current configuration, then rewrites that configuration in the manner that the rule specifies. In the occasion that the preconditions for multiple rules are matched, the runtime can concurrently apply many rules at once, as long as no rule modifies a term that another rule matches.

4.3 Annotations

In Chapter 3 we mentioned *syntax annotations*. When one defines the syntax of a language, we can add these annotations to the CFG productions that comprise the syntax definition. These signal certain behaviours to the runtime, allowing us to obtain certain behaviours "for free", without having to write rewrite rules for them. This is helpful for common behaviours that occur across many different expressions. We can also add annotations to some rules to specify certain behaviours as to how they should be applied.

4.3.1 Strictness

When a subexpression of an expression in a production rule is marked as strict, this tells the runtime to prioritise evaluating rules on the subexpression so until it is evaluated to something that an explicit rewrite rule can work on. For example:

Snippet 4 This syntax declaration and rewrite rule give the same effect as the notation in Snippet 2

```
syntax Expr ::= Int
              | Expr + Expr [strict]
rule I1:Int + I2:Int => I1 +Int I2
```

The left hand side and right hand sides of the addition expression have been marked as *strict*. This instructs the runtime to extract the expression and place it at the head of the program. Then, various other rewrite rules will be applied that we have hopefully defined earlier. Then, once some rewrite rules are applied, the new expression is placed back in the superexpression that its predecessor was a subexpression of. This is the equivalent to the first two rules of the form $\frac{e_0 \to e_0'}{e_0+e_1 \to e_0'+e_1}$: we evaluate e to some unknown e', and replace e with it. Finally, there will come a point where there exists no such e', as no rewrite rule will match e, so the runtime will try to evaluate the next subexpression.

Once all subexpressions have been evaluated, then they are literals (or the specification is incomplete). Then, the single rule we have defined, that rewrites an addition expression to the algebraic sum of the two literals, is applied. This is the same as the third rule in Snippet 2, which works on literals. The other two rules existed to convert expressions into literals, but we have obtained this behaviour implicitly by giving strictness annotations to the syntax elements.

If we were to implement this behaviour in rewrite rules, it would result in a lot of boilerplate, as this behaviour would take several rules (for each subexpression, we need a rule to move the subexpression out, and a rule to move the new subexpression in), for each and every expression in the language. Thus, annotating strictness in the syntax defintion saves us having to write these rules and instead focus on writing the rules that define unique behaviour.

4.3.2 Other annotations

`[strict]` is not the only possible syntax annotation; there are many different common ”boilerplate” behaviours that we would like to abstract away rather than writing rewrite rules for. For example, we can specify an index or indices (i.e `[strict(1,3)]`) to specify that only the subexpressions at those indices should be evaluated strictly; we may want to do this so that we can apply an unevaluated expression (i.e not a literal) onto expressions that must be evaluated into a literal beforehand. Or, we may just have our own notion of how we wish to treat subexpressions that is just not captured by strictness, and therefore want to leave it out and define our own rules.

Other syntax annotations include `[left]` and `[right]`, which specify that a given production is left- or right- associative. Another one that I used often is `[function]`, which signifies that a given production should be prioritised in evaluation; it is not supposed to be a term that is manipulated or passed around as is, but rather to be evaluated immediately. Similarly, there is the `[functional]` label, which is useful in conjunction with `[function]`. This tells the runtime that not only is the production a function that must be immediately evaluated, but the evaluation is deterministic; the same expressions as ”arguments” will lead to the same result.

Another annotation used in my implementation is `[structural]`. This annotates a rule rather than a syntax element, and is used to denote rules that are not computational steps. For example, we may label rules that ”clean up” local environments as structural; they are not really computations.

4.4 Previous Work

Various languages have been implemented as $\mathbb{K}$ specifications, such as C [7], python [4], and Haskell [5]. However, these languages are all examples of ”traditional” languages, rather than esoteric programming languages, let alone fungeoids. As such, it would be interesting to implement an esoteric language and see how naturally K would handle it, as a form of ”stress test”. Because Piet is such an unusual language, it is a good candidate for this test.

While doing research on previous solutions, I found a set of $\mathbb{K}$ specifications for esoteric languages [8], including a specification for `befunge93`; the language for which fungeoids are named. However, this specification was out of date; it would not work with newer versions of the K framework. In addition, Piet is rather different from Befunge; in Piet, the program pointer is moved between blocks of same-coloured pixels, each of which is considered a single unit, but is comprised of many pixels, while in

Befunge, each individual cell (i.e position in space) is a complete unit. In spite of these issues, certain parts of this specification are applicable to an implementation of Piet.

For example, the `befunge` implementation's configuration stores a mapping between position in space and the symbol at that position. I reuse this idea in KPIET by using such a mapping cell that maps from a position in the image to a unique identifier for the colour block that pixel belongs to. In addition, the Befunge implementation uses a Virtual Machine architecture, which I copied. These will be covered in more detail in Section 5.

Chapter 5

Implementing Piet

When implementing a language it is important to break the work down into several components that can be each implemented separately. Each subsystem is simpler than the whole, and is thus easier to implement correctly. In addition, subsystems can be encapsulated and thus be agnostic about the internal function of other components. This means that should a subsystem be found to be incorrect, we can fix the subsystem without having to worry about the effects on other subsystems, as they are not tightly coupled.

5.1 Bitmapping and Parsing

The first problem an implementation of Piet will encounter is how to load an image. Over the decades, a great deal many image encoding formats have been devised, usually for the purpose of compression, such as the PNG[3] and JPEG standards[21], but Piet works on a basis that the colour of every pixel is known. Thus, any image would have to be converted to an uncompressed bitmap format for the specification to be able to work on it - such a format does not need any further word to unencode individual pixel values, and Piet works on the assumption that each pixel value is known.

A naive solution would be to implement semantic rules to convert arbitrary input into its corresponding bitmap interpretation. However, modern image formats are complex enough that to define their ”semantics” would be a project of a scope at least as large as this one, not to mention the problem of figuring out which format is being used. Secondly, the semantics of image formats are not the subject of the project, despite likely being a valuable undertaking for practical purposes.

In fact, a similar problem was present in the K specification for `Befunge93` [17], which is a language that stores instructions in a two-dimensional grid, like Piet. The solution taken here was to create a separate program that converted the input program string to a mapping of positions in space to cell contents. Then, a script would ”inject” this representation as a cell when the K interpreter was called, thus avoiding the need for parser rules within the specification, which do not technically constitute the semantics of the program.

With this solution in mind, I wrote a simple program that loaded the program image, converted it to a bitmap, and wrote the colour of each pixel in hex representation to file. This was written using the `Rust` [9] language; the language claims good support for writing simple command-line scripts like this, has an image-processing library, and I wanted to practise programming in it. This does create a dependency on the Rust toolchain (`cargo`) if one intends to convert images to the intermediate file format.

This file is the input for the K specification. Unlike the Befunge example, the bitmapping program does not inject the parsed image as a cell into the K runtime. This quality would be desirable, as it would make the pipeline a single process, but instead the file is read as an input program for the K specification. This was because injection is an advanced feature, and not a well-documented one, so I felt it simpler to create a small number of rules to parse the program into the appropriate cell mapping positions to pixels. (Appendix) On the other hand, this intermediate representation meant that I did not have to repeat the bitmapping process every time I wanted to test the semantics, I could read in the bitmapped files rather than images.

Snippet 5 The rules responsible for parsing the intermediate bitmap representation into the position-to-colour mapping. Note that the `<k>` cell holds the current state of the unparsed IR, while `<program>` stores the mapping.

```
rule [finished-parsing]:
    ⟨k⟩.Lines => step ⟨/k⟩

rule [parse-next-line]:
    ⟨k⟩L:Line ; Ls:Lines => L ⟨/k⟩
    ⟨nextLines⟩ . => Ls ⟨/nextLines⟩
    ⟨buildingx⟩ _X:Int => -1 ⟨/buildingx⟩
    ⟨buildingy⟩ Y:Int => Y +Int 1 ⟨/buildingy⟩ [structural]

rule [parse-next-line-restore]:
    ⟨k⟩ .Line => Ls ⟨/k⟩
    ⟨nextLines⟩ Ls:Lines => .⟨/nextLines⟩ [structural]

rule [parse-next-pixel]:
    ⟨k⟩P:Pixel  L:Line => P ~> L⟨/k⟩
    ⟨buildingx⟩ X:Int => X +Int 1 ⟨/buildingx⟩ [structural]

//place the colour index into the program cell, mapped from its position.
//This means we will be able to look up colours from positions later
rule [place-pixel-in-map]:
    ⟨k⟩ C:Colour => . ⋯⟨/k⟩
    ⟨buildingx⟩ X:Int ⟨/buildingx⟩
    ⟨buildingy⟩ Y:Int ⟨/buildingy⟩
    ⟨program⟩ ... .Map => point(X,Y) |-> C ⋯⟨/program⟩ [structural]
```

Note in the above listing that an image is a collection of `Pixel` objects, but the `<program>` cell maps to `Colour` objects. This is because there is a set of rules that convert any seen `Pixel` object into the lightness/hue representation of `Pixel`. This implicitly occurs between the execution of `parse-next-pixel` and `place-pixel-in-map`. An interest is that colours not explicitly named in the specification are treated as implementation-

specific behaviour. However, it does recommend taking the interpretation that they be treated as white pixels. To this end, there is a rule that matches the case of an unrecognised colour, and converts it to white. (Appendix, `translate-hexcode`)

Snippet 6 Converting a non-specification colour into a white pixel. Note that a similar rule exists for converting a hexcode into the colours that are part of the specification.

```
rule [translate-hexcode-encountered-illegal]:
TranslateHexcode _:Id => color ( white )
```

5.2 Virtual machine

The next part of the implementation was to create a virtual machine for the execution of instructions. Such an approach is used in most existing implementations of Piet, as transitions between colours are translated into instructions affecting the stack. As such, a virtual machine architecture, that takes a single simple instruction at a time and applies it to the program's state, is a natural fit.

5.2.1 Executing Instructions

The creation of the virtual machine's execution was straightforward; it was not a feature unique to Piet. The execution of any given instruction is decoupled from the rest of the semantics of Piet; indeed, they could be used as the basis for a simple conventional language, with the exception of the `pointer` and `switch` instructions. As such, these were implemented first, as they were unlikely to need to be rewritten later.

Each instruction corresponds to one or two short rules that modify the stack. This makes it very clear as to what each instruction each semantic rule corresponds. This clear correspondence was of great benefit when using the Literate Programming solution, detailed later. The simpler rules, like addition, were implemented as singular atomic state transitions, that match some number of elements off the stack, and rewrite them into the output element, in this case the sum.

Snippet 7 The "Duplicate" rule. This rule rewrites the top item on the stack into two copies of itself, then schedules a no-op.

```
rule [instruction-duplicate]:
    ⟨k⟩ dup => nop⟨/k⟩
    ⟨stack⟩ListItem(Value:Int) => ListItem(Value) ListItem(Value) ⋯⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("DUP") ⟨/log⟩
```

Other, more complex rules required the breaking down of the instruction into several simpler ones. A good example here is the case of the "roll" function.(Appendix, `instruction-roll`) This function takes the top two numbers off the stack, and buries the top element down the stack a number of elements equal to the second value, a number of times equal to the first. This is far too complex for a single rewrite rule as

there are multiple parameters for the rule. Instead, the roll function is rewritten as an intermediate function, where it ”counts down” from the repetition value, doing a single burying operation at a time.

Snippet 8 The ”Roll” rule, and (part of) the supporting rules.(Appendix)

```
rule [instruction-roll]:
    ⟨k⟩roll => rollby(Depth, NumRolls)⟨/k⟩
    ⟨stack⟩ ListItem(NumRolls:Int) ListItem(Depth:Int) => .List ... ⟨/stack⟩
        requires Depth ⟩=Int 0

rule [instruction-roll-negative-depth]:
    ⟨k⟩roll => nop⟨/k⟩
    ⟨stack⟩ ListItem(_:Int) ListItem(Depth:Int) => .List ... ⟨/stack⟩
        requires 0 >Int Depth //negative depth is an error and ignored

rule    ⟨k⟩rollby(_:Int, 0) => nop⟨/k⟩

rule    ⟨k⟩rollby(Depth:Int, NumRolls:Int) => rollby(Depth, NumRolls -Int 1)⟨/k⟩
        ⟨stack⟩
            S:List =>
            (
                range(S, 1, size(S) -Int Depth )
                range(S, 0, size(S) -Int 1)
                range(S, Depth,0)
            )
        ⟨/stack⟩
            requires NumRolls >Int 0
```

More importantly, these rules only modify the stack for the most part. This means that the surface of interaction between the VM and the rules that would be implemented later, such as program flow, was small and the systems were thus decoupled. This meant that these rules did not have to be amended when changed were made to the design of the flow control rules.

One aspect of Piet is that, when an instruction cannot be executed, it is ignored. This usually arises if the stack contains less elements that the required number of operands of the instruction. To solve this, instructions were placed in different syntax categories, rules were added to match these categories of instructions in the situation where not enough stack elements were present. (Appendix, `ignore-one-arg-instruction`, `ignore-two-arg-instruction`) This reduced code reuse of a similar rule for every individual instruction. In addition, K’s semantics make this ignoring very easy; the only state transition is that of the instruction to a no-op.

All instructions execute a no-op after execution or ignoring. This is the exit point for the virtual machine; the execution of this no-op puts the program back into the state where it is about to take a ”step”; i.e find the next instruction to execute.(Appendix, `process-nop`) There is also a special no-op, (Appendix, `process-nop-loop`), that is executed when moving between white blocks to make sure the program is in a loop.

Snippet 9 The rules that allow us to ignore instructions with insufficient arguments. Note that `Instruction2Arg` is a syntax category that comprises the identifiers of the instructions that require two arguments to function. `Instruction1Arg` comprises the identifiers of instructions that require only one argument.

```
rule [ignore-one-arg-instruction]:
    ⟨k⟩ _:Instruction1Arg => nop ⟨/k⟩
    ⟨stack⟩ S:List ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("IGNORE,") ⟨/log⟩
    requires 1 >Int size(S)

rule [ignore-two-arg-instruction]:
    ⟨k⟩ _:Instruction2Arg => nop ⟨/k⟩
    ⟨stack⟩ S:List ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("IGNORE,") ⟨/log⟩
    requires 2 >Int size(S)
```

5.2.2 Parsing Instructions

The other half of the virtual machine was the front end of the VM; which would take the two colours the program had moved between and produce a VM instruction for the execution rules to handle. This was further split into rules to find the differences in hue and lightness.

Whenever the program pointer transitioned between two different colour blocks, the specification would call a rule that calculates the differences in hues Appendix, `hue-difference-` family of rules) and lightnesses (Appendix, `lightness-difference-` family of rules) of two colours, then call a second rule that would look these differences up and translate to a VM instruction (Appendix, `instruction-resolution-` family of rules). These rules thus act as the entrypoint from the program flow rules to the virtual machine rules.

As part of these rules, I had to devise a representation for colours. Most implementations store a colour as an integer, however, I instead decided to store colours as a set of two named tokens representing the hue and lightness. This allowed the lookup to be split into two separate functions for finding the hue and lightness differences. Secondly, because colours were stored as named values, the specification became easier to debug as one could see what colours were being considered rather than numbers that had to be translated into names.

This required defining the meaning of equality between two colours and their components. However, the exact semantics of such functions are not intrinsic to Piet, and as such I decided to move them into a separate file of helper functions that is imported by the specification. This should help understandability since there are less of these helper functions to clutter up the important rules in the specification.

5.3 Program flow

The way in which program flow works is what makes fungeoids unique, and as such is the most important part of the implementation.

In Piet, as with other fungeoids, there is a ”direction pointer” which points in the direction in which the program executes. Unlike other fungeoids, however, Piet’s execution operates on the basis of moving between blocks of similarly-coloured pixels, in the aforementioned direction, rather than between individual pixels. Secondly, black and white pixels have a special function; the direction pointer ”bounces” clockwise if the program pointer moves to a black pixel, in effect functioning as a ”wall” to direct program flow. White pixels on the other hand are ”empty”, the program pointer can move through them without executing instructions.

As part of this program flow, one must find the borders of a colour block, so that one can find the next block to move to. In most implementations, these two problems are solved in one step. Some implementations precompute blocks, but they are the exceptions, like repiet. This is possibly a function of the majority of implementations being interpreters rather than compilers; compilers by definition must precompute blocks. However, for ease of understanding, and for decoupling of functions, a single rule for block transitions was created as in the latter case.

In this rule (Appendix, `next-step-mapping-exists`), we look up the block the current pixel belonged to, and look up the block definition to see where the next pixel to move to would be, given the current state of the direction pointer. This assumes that the blocks have been precomputed. The reason for this choice is that the specifics of how this block and its boundaries are computed is just an implementation detail, and by doing this we decouple the program flow from this calculation. This allowed me to try several solutions to computing blocks. Secondly, this meant that the transition rule was implemented as a single functional rule, which mirrors the human-language specification, which makes no demand as to how the blocks are to be computed.

However, it is not necessarily the case that the block has been computed already. If this is the case, the transition rule cannot be matched. Instead, the block computation rule is matched (Appendix, `next-step-no-mapping`), which calls an expanding-frontier method (Appendix, `build-block`) and then queues up another step of execution, in which the normal transition rule should be matched. The expanding frontier method adds pixels it knows are of the same colour as the colour of the current position of execution; in other words, the colour of the currently needed block. Then, once it finds all these pixels and maps them to their appropriate block ID label, another set of rules (Appendix, `build-block-<direction><direction>` family of rules) finds the pixels at the most extreme edges of the block. These are the positions that the transition rule needs to find. After all this process, the program has returned in the same state it was before the block-building process was called, with the one difference that a new block has been computed. This should allow the transition rule to properly apply and move to the next step of execution.

There are of course, a few exceptions to the transition rule. This occurs in the case of entering black or white pixels.

Snippet 10 This rule captures the basic movement semantics of Piet. If this rule is not matched, and the target coordinate is within bounds, the runtime will match a different rule that attempts to find the boundaries of the block to which the pixel at the current position belongs to. This is then stored in a `block` cell, which this rule can then match and continue executing the program.

```
rule [next-step-mapping-exists]:
    ⟨k⟩ step => TranslateInstruction OldColour NewColour ... ⟨/k⟩
    ⟨PP⟩ OldPP:Coord => NewPP +Coord DPToOffset(D) ⟨/PP⟩
    ⟨exitedPP⟩ _ => OldPP ⟨/exitedPP⟩
    ⟨program⟩ ... (NewPP +Coord DPToOffset(D)) |-> NewColour:Colour ⋯⟨/program⟩
    ⟨owner⟩ ... OldPP |-> OldBlockID:Int ... ⟨/owner⟩
    ⟨blocks⟩
    ...
        ⟨block⟩
            ⟨id⟩ OldBlockID ⟨/id⟩
            ⟨colour⟩ OldColour:Colour ⟨/colour⟩
            ⟨size⟩ _Size:Int ⟨/size⟩
            ⟨transitions⟩ ... direction(D,C) |-> NewPP:Coord ... ⟨/transitions⟩
        ⟨/block⟩
    ...
    ⟨/blocks⟩
    ⟨DP⟩D:DirectionPointer⟨/DP⟩
    ⟨CC⟩C:CodelChooser⟨/CC⟩
    ⟨blockworkspace⟩ .List ⟨/blockworkspace⟩
```

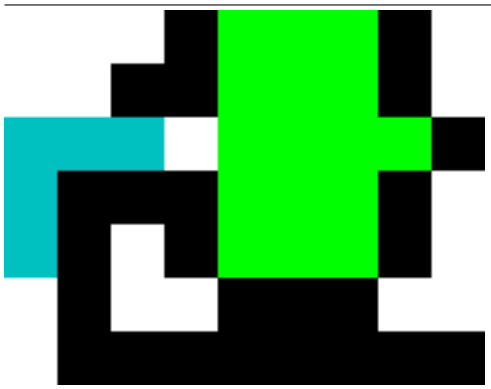
Black pixels Piet uses black pixels as ”walls”; program flow cannot enter them, but instead are ”bounced” off of them, allowing the programmer to use them to control program flow. To implement this, I wrote a rule that moves the direction pointers whenever the colour of the newly-entered block is black, and steps the program back to the position it was in before it moved into the black block. (Appendix, `hit-black-pixel-dp`). To do this; I needed to find the first place such a case can be detected; this was in the translation function that converted colour transitions to instructions. (Appendix, `translate-instruction-to-black`). The back-stepping required the specification to keep track of the previous position; but this is just a case of adding a configuration cell and matching it in the appropriate rules, it did not require the creation of any new rules.

The other important function of black pixels is that if one collides with them too many times without execution an instruction in between, the program halts. This allows the programmer to terminate the program by creating a ”mushroom”-shaped block surrounded by black pixels. To solve this problem, I extended the rules rotating the direction pointers to check that the number of times this had happened did not exceed 8: if it did, then these rules were not matched, and instead a different one, that rewrote the program state to halt was matched. (Appendix, `hit-black-pixel-too-many`) Then, the no-op rule was modified to reset this counter, because a no-op is called after finishing movement into any non-black pixel.

Snippet 11 The rule responsible for termination when one cannot exit a color block. The argument in `blk()` is the number of times one has hit a black pixel without the counter being reset by finding a valid move; if this has occur ed 7 times then we conclude there is no valid way to exit the block.

```
rule [hit-black-pixel-too-many]:
    ⟨k⟩blk(7)  => stop …⟨/k⟩
    ⟨log⟩ ... .List  => ListItem("stuck") ⟨/log⟩
```

Snippet 12 A ”mushroom” shaped block. Here, when the program enters the green block, all valid moves out of the block would enter the black border or the image borders at the top. The program cannot leave the block and terminates.



White pixels The other colour with special semantics is white. White pixels function as empty space; execution steps that move to or from a white pixel do not execute a VM instruction. This is represented by adding rules to the colour-transition-to-instruction function that match the cases where one or more of the colours is white (Appendix, `translate-instruction-from-white`, `translate-instruction-to-white`, `translate-instruction-between-white`), and translate to the no-op instruction.

The second feature of white pixels is that they do not form blocks like other coloured pixels. Instead, the program pointer ”slides” through white pixels in the direction of program flow, bouncing off black pixels, until it reaches a coloured pixel. This effect can be adapted from the normal rule of moving between blocks in the direction of the program pointer by making sure that white blocks are never larger than a single pixel in size. For this purpose, a rule was created (Appendix A, `build-block-white`) that would be matched instead of the normal block-building rule. This rule did not use the frontier expansion method, and instead created a block containing only the pixel that would have otherwise been the starting point for the frontier.

The final special feature of white pixels is loop termination. If the program is stuck in a loop of white pixels, it cannot exit, as no conditions will change. Therefore, the specification states that in such a condition the program must terminate. If there are any coloured pixels in this loop, then this does not apply. This means that the program must check every execution step between white pixels that it is not in a loop. To make this check, the instruction translation rule was amended to include a case for moving between two white pixels (Appendix, `translate-instruction-between-white`).

This would go to a special no-op case (Appendix, `process-nop-loop`). This no-op checks that the program is not a loop, and, if it is in a loop, halts the program, and otherwise rewrites to the standard no-op rule and continues execution as normal.

This looping check created many problems during development due to its complexity. The conventional [20] approach to infinite loop detection involves running a second copy of the program, more slowly than the first, and checking if the two converge to the same state at some point in time. The idea here is that if the state is the same at two different points in time, then program state must be periodic, with a period equal to the difference in time step between the two programs. This is known as a ”Tortoise and Hare” technique. This was originally used in PROLOG interpreters, to detect if the interpreter was stuck in a loop repeatedly deriving the same rules. In theory, I could have implemented this solution, however, I felt that it was complicated and was not confident I could implement it it a reasonable timeframe.

The solution that was settled upon required creating a function that would take the last n positions the program pointer had taken, verify that the pixels at these positions were white - otherwise, the program was not in a loop - and then verify that the sequence of the previous n positions was equal to the first set of positions. If so, then the program is in a loop. If the program was still not determined to be in a loop, the function was called again with the value of n incremented until n was equal to half of the length of the sequence of positions. This system catches such infinite loops, as proved by testing. To make this solution work, I had to amend the no-op VM rule to also log the position of the program pointer; this was the logical choice as this rule is called after every movement of the program pointer, but not when bouncing off of black walls. This log of positions was stored in a cell that was fed into the loop checking function. An upshot of this was that this allowed for easy debugging of the program, as this meant there exists an inspectable log of program flow.

5.4 Literate Programming

Literate Programming [11] is a term used to talk about programs that are presented as their human-language specification annotated with the relevant code. It makes sense to talk about language specifications in this way, with a human-language rule annotated with its K equivalent.

There are many ways to implement this; for example, compiled definitions can create a PDF file as output that interleaves the specification and code, or a webpage, or the code is embedded within a text document. Interactive Notebooks like Jupyter can be seen as an example of the latter, where blocks of code are interspersed with paragraphs of text. In such a workbook, the text is given priority, as it explains the concepts and meanings that the author intends to impart upon the reader, with the code a supplementary part to aid in the understanding of the text. In contrast, in a source code file the majority of the file is code, with small amounts of text to explain the code.

Conversely, the literate programming paradigm can also be used to disambiguate the meaning of code; because the core of the file is the specification, the purpose of the code, which is secondary, is made clear. This means that extensive commentary is

no longer necessary, as a well-written specification should make clear the intended function of any block of code.

The K framework has support for a form of Literate Programming; the `kompile` tool accepts Markdown files as input. These Markdown files can contain code blocks marked as K code; when the file is passed to `kompile`, the sections not in such code blocks are discarded as non-functional text. This means that the a semantic rule can be placed next to the part of the specification it implements, as an annotation. Taken over a whole specification, this allows us to write a specification in a human language, which is easier to read, and annotate this text with executable semantic rules, disambiguating the meaning of the text.

This is different from Knuths original `WEB` implementation of a Literate Programming environment, which works on a system of structured macro definitions, which can contain commentary as well as code, and can call other macros, in contrast to $\mathbb{K}$'s "annotated source file" format. However, the similarity between this system and $\mathbb{K}$'s Literate Programming facilities arise as the parallels with the "weave" and "tangle" functions; where "weave" in `WEB` converts the program into a TEXdocument, the Markdown nature of the KPIET specification allows me to convert it into a presentable format such as a webpage (or a LATEXdocument, which is how the Appendix was formed). On the other hand, the "tangle" function takes only the code sections of the `WEB` document, and expands the macros until it options a complete program. $\mathbb{K}$'s `kompile` tool achieves a similar goal by extracting all code from the relevant code blocks and concatenating them to create the source code to create an interpreter from.

With this capability in mind, I decided at the beginning of the project to produce such a system; at the time I started learning it, K supported a notation format for source code, that when passed to kompile, would create a pdf file using the explanations and with the semantic rules in a graphical format. However, this feature was deprecated soon after. However, the markdown solution outlined above was implemented quickly. As such, when the project neared completion I made a copy of the original Piet specification I was working off of, converted it to a Markdown format, and then copied across the K semantics that had been written already into code blocks as annotations.

Chapter 6

Evaluation of the specification

6.1 Constructing a test suite

From the beginning, it was decided that the first task to complete should be the creation of a test suite that would test each aspect of the language. These tests were pulled from various sources, mostly David Morgan-Mar's page of example programs. [15] In fact, the "Hello World" example alone has an example of every virtual machine instruction. As such, using several examples from this page gave near-complete coverage for all features of the language.

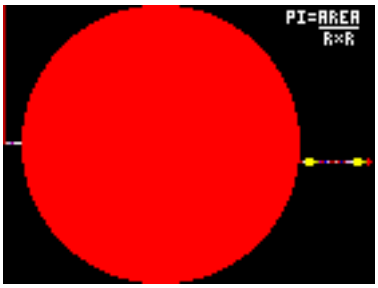
However, this set of examples did not have examples of programs that tested every aspect of the program flow routing features of black and white pixels. This meant I had to create some examples of programs to test that the direction pointers were correctly modified when bouncing off of black pixels, that white pixels were moved through correctly, and that the program exited when in a loop. To create the first two, I created a program (`pathing.gif`, 6.1,left) with a path of white pixels on a black background that, if followed correctly, would lead to the entrypoint of the "Hello World" program. Thus, if the program display "Hello World", one could conclude that the test passed. In the latter case, I created a program with a loop in white space (`loopy.gif`, Figure 6.1,right), it would be expected to terminate. Furthermore, I used a test case from an existing Piet interpreter [2] of a program that would never terminate, due to the loop having a coloured pixel in it. The last program I created was a variant of the pathing test, where the white space was replaced with colours that were not in the specification; these are to be treated as white pixels (`illegal_colour.gif`, 6.1).

Running each of these tests by hand (and remembering the expected output) would be a time-consuming exercise, so I wrote an automated testing system, using a shell script. Using this system, I created a set of folders, each containing a single program, and a series of expected input and output files. The shell script goes through each folder, and runs the program in that folder with each present input file piped in as standard input. The output was then compared using the `diff` command to the corresponding output file of the same name as the input file. In this way I could write several sample inputs and outputs as test cases for each program. This allowed me to quickly test all the test cases while developing the specification. However, the primitive nature of this

Figure 6.1: The two images used to test the program flow. The left checks that the direction pointers ”bounce” properly when colliding with walls, until the program reaches the coloured blocks where it prints ”Hello World”. The right program tests the loop detector; the program should enter a loop in the white area and never enter the coloured blocks. The program should terminate with no output; seeing output from the coloured pixels would imply that there is something wrong with the pathfinding.

Figure 6.2: Test case used to test illegal colours. The interpreter should treat this in the same way as `pathing.gif` in 6.1, left, because the pink and olive pixels are unspecified and thus treated as white.

Figure 6.3: The Pi calculator test case; this image is 139 by 104 pixels in size, the largest test case. [13]



testing system did not account for the test that would run indefinitely, and thus I had to exclude this specific test from the system and run it manually.

6.2 Results of testing

As of writing, all but one test in the suite pass, which suggests that the implementation correctly captures the semantics of Piet. This single failing test is one testing the integer input VM instruction; specifically, it crashes the program if given improper input. The reason for this is that by its definition STDIN cannot be read without consuming. Indeed, there is no clear consensus amongst existing implementations as to how to handle this; some implementations do not consider the possibility of non-integer input and crash, while some consume and discard the entire input, even if the prefix of the input can be interpreted as a valid input. Others still consume the entire buffer instead of just the first character. This seems to imply that the specification around this operation is ambiguous.

In addition, the Pi Calculator test case [13] (reproduced below) crashes by running out of memory before any semantic rule is applied by the runtime. This issue seems to stem from the size of the program; it is the largest of all the test programs, and redrawing the program at a lower resolution - an interesting artefact of this specific program, in that it can calculate more accurate values of pi at higher resolutions, but still works at lower ones - allows it to run correctly. This suggests that the problem is either due to the parser of the syntax of the bitmap input files being inefficient and using too much memory, or with the K runtime itself. The former seems more likely, as K can handle large programs in more conventional programming languages. I do not feel this is a disqualifying issue, because the subject of this project is the semantics, and the syntax issue can likely be solved by using the injecting mechanism, with no change to the semantic rules.

6.3 Improvements

While the solution seems to work, and thus demonstrates the viability of denoting fungeiod semantics in K, there still remain a few places in which it can be improved or extended.

Probably the most obvious improvement that would result in the greatest benefit would be to implement the cell injector as detailed in the Program Parsing section; this comes with several benefits. Firstly, it would make the whole system self-contained with no intermediate representation files created as is the case currently. Secondly, it would allow for the removal of the program parsing rules, which I find to be particularly troublesome. On the most conceptual level, they deal with the *syntax* of Piet rather than the semantics, and the project itself focuses on the latter. Not to mention, this is using semantic rules to implement syntax, but these should ideally be kept seperate; we are not concerned with the behaviour of the parser. Parsers are well-studied and are not a component of the behaviour of the languages, rather a component of the behaviour of a specific implementation of the language. On a more practical level, these rules have performance issues; on the small test programs, they account for several hundred execution steps, in some cases dwarfing the number of execution steps taken by the program itself, and this results in a large delay between starting the program and getting a response from it. In addition, when large programs are executed, the K runtime crashes in its parsing step; if the program were injected into the storage pre-parsed, this would probably not occur.

In addition, the bug with the behaviour of the integer input as detailed in the Testing section above is obviously an area where the implementation can be improved. However, given that other implementations have trouble implementing this rule, this may be a case where the specification is unclear as to what it wants; for example, how should one handle the input ”123 abc”, as the integer input 123 followed by the string abc in the next input operation, or as a single string? And what about ”123abc”?

One feature that I purposefully ignored from the beginning was the idea of variable-sized ”codels” - square areas of colour, the size of which could be specified when the program was executed; instead, I assumed the codels were of size 1 and thus lined up with the real pixels of the image. However, this feature should not be difficult to implement, and could be implemented as part of the front-end program; instead of writing the colour of each pixel, it would output the colour of each n by n block of pixels, with n specified at runtime.

Finally, expanding the test suite would be a good improvement; while I am convinced that the test suite covers all the semantics of Piet, adding more tests, particularly testing the pathing rules, would help in assuring others that this is the case.

Chapter 7

Conclusions

We have successfully implemented Piet as a rewrite rule system. The specification of this rule system is reasonably understandable, so we can conclude that the $\mathbb{K}$ framework is general enough to represent the semantics of fungeoids in an intuitive manner. This is useful for proponents of $\mathbb{K}$, who can use this to argue the capabilities of $\mathbb{K}$'s rewrite logic. It can also be used as a pedagogical too to teach reasoning in $\mathbb{K}$ without being constrained by a specific programming paradigm, by exposure to a variety of paradigms.

As mentioned earlier, there is still the problem of how to disambiguate ignoring bad input on the integer read function. This is possibly due to the original description being in human language; once a definition is formally specified, this no longer need be a problem for any hypothetical implementation as the formal specification can be referenced. Along the same line of thought, the specification in general can be seen as a reference complementing the original natural-language specification.

In future, it may be of value to continue to study other unusual language paradigms, such as a cellular automata system, to see how well these can be represented as rewrite systems.

Bibliography

[1] Kelly Boothby. Repiet: The brutalizing piet recompiler. https://github.com/boothby/repiet, 2019. Accessed 2021-02-28.

[2] Jens Bouman. Piet interpreter. https://github.com/JensBouman/Piet_interpreter, 2020. Accessed 2021-02-27.

[3] W3C comittee. Portable network graphics (png) specification. https://www.w3.org/TR/PNG/, 2003. Accessed 2021-04-06.

[4] K Framework Contributors. python-semantics. https://github.com/kframework/python-semantics, 2013. Accessed 2021-04-06.

[5] K Framework Contributors. Ghc core in k. https://github.com/kframework/haskell-core-semantics, 2016. Accessed 2021-04-06.

[6] K Framework Contributors. K. https://kframework.org/index.html, 2020. Accessed 2021-03-24.

[7] K Framework Contributors. c-semantics. https://github.com/kframework/c-semantics, 2021. Accessed 2021-04-06.

[8] Chucky Ellison. esolang-semantics. https://github.com/ellisonch/esolang-semantics, 2015. Accessed 2021-03-27.

[9] Rust Foundation. Rust programming language. https://www.rust-lang.org/, 2021. Accessed 2021-02-26.

[10] Hans Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010.

[11] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 01 1984.

[12] Nimrod Libman. Kpiet. https://github.com/ZayadNimrod/KPIET, 2020. Accessed 2021-04-09.

[13] Richard Mitton. Pi. https://www.dangermouse.net/esoteric/piet/samples.html, Unknown date. Accessed 2021-03-19.

[14] David Morgan-mar. Dm's esoteric programming languages - piet. https://www.dangermouse.net/esoteric/piet.html, 2018. Accessed 2021-04-07.

[15] David Morgan-mar. Dm's esoteric programming languages - piet samples. `https://www.dangermouse.net/esoteric/piet/samples.html`, 2020. Accessed 2021-04-09.

[16] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[17] Chris Pressey. Befunge93 documentation. `http://www.nsl.com/papers/befunge93/befunge93.htm`, 1993. Accessed 2021-02-26.

[18] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[19] Erik Schoenfelder. npiet - an interpreter and editor for the piet programming language. `https://www.bertnase.de/npiet/`, 2020. Accessed 2021-02-28.

[20] Allen Van Gelder. Efficient loop detection in prolog using the tortoise-and-hare technique. *The Journal of Logic Programming*, 4(1):23–31, 1987.

[21] Joan L. Mitchell William B. Pennebaker. *JPEG:Still Image Data Compression Standard*. Springer US.

Appendix A

The Specification: kpiet.md

The main specification file is reproduced below. The complete codebase, including a file of defined ”helper” functions, can be found in the project repository.[12]

Piet

```
require "helpers.k"

module KPIET
    imports DOMAINS-SYNTAX
    imports DOMAINS

    imports HELPERS
```

Language Concepts

Colours

Piet uses 20 distinct colours, as shown in the table underneath. The 18 colours in the first 3 rows of the table are related cyclically in the following two ways:

- **Hue Cycle:** red -> yellow -> green -> cyan -> blue -> magenta -> red
- **Lightness Cycle:** light -> normal -> dark -> light

Note that ”light” is considered to be one step ”darker” than ”dark”, and vice versa. White and black do not fall into either cycle.

```
//The difference between two lightnesses of the same lightness is zero
rule [lightness-difference-base-light]:
    LightnessDifference L1:Lightness   L2:Lightness
     => 0  requires L1 ==Lightness L2

//if the colours are dissimlair, darken the second lightness one step,
//check the difference on this new lightness, and add 1 to it
rule [lightness-difference-inductive-light]:
    LightnessDifference L:Lightness       normal
     => LightnessDifference L light +Int 1 requires notBool(L ==Lightness normal)
rule [lightness-difference-inductive-normal]:
    LightnessDifference L:Lightness       dark
     => LightnessDifference L normal +Int 1 requires notBool(L ==Lightness dark)
rule [lightness-difference-inductive-dark]:
    LightnessDifference L:Lightness       light
     => LightnessDifference L dark +Int 1 requires notBool(L ==Lightness light)


//The difference in hues of the same hues is zero
rule [hue-difference-base-red]:
    HueDifference       H1:Hue      H2:Hue      => 0 requires H1 ==Hue H2

//If the colours are dissimiliar, then step the second hue down,
//then check the hue difference between those two colours, and add 1 to it
rule [hue-difference-inductive-yellow]:
    HueDifference       H:Hue       yellow
     => HueDifference H red      +Int 1 requires notBool(H ==Hue yellow)
rule [hue-difference-inductive-green]:
    HueDifference       H:Hue       green
     => HueDifference H yellow   +Int 1 requires notBool(H ==Hue green)
rule [hue-difference-inductive-cyan]:
    HueDifference       H:Hue       cyan
```

```
     => HueDifference H green    +Int 1 requires notBool(H ==Hue cyan)
rule [hue-difference-inductive-blue]:
    HueDifference       H:Hue       blue
     => HueDifference H cyan     +Int 1 requires notBool(H ==Hue blue)
rule [hue-difference-inductive-magenta]:
    HueDifference       H:Hue       magenta
     => HueDifference H blue     +Int 1 requires notBool(H ==Hue magenta)
rule [hue-difference-inductive-red]:
    HueDifference       H:Hue       red
     => HueDifference H magenta  +Int 1 requires notBool(H ==Hue red)
```

#FFC0C0 light red	#FFFFC0 light yellow	#C0FFC0 light green	#C0FFFF light cyan	#C0C0FF light blue	#FFC0FF light magenta
#FF0000 red	#FFFF00 yellow	#00FF00 green	#00FFFF cyan	#0000FF blue	#FF00FF magenta
#C00000 dark red	#C0C000 dark yellow	#00C000 dark green	#00C0C0 dark cyan	#0000C0 dark blue	#C000C0 dark magenta
#FFFFFF white			#000000 black		

```
syntax Colour ::= "TranslateHexcode" Hexcode [function]

//we always want to convert from hex whenever we can
rule H:Hexcode => TranslateHexcode H      [structural]

rule [translate-hexcode-encountered-hexcode-light-red]:
    TranslateHexcode xffc0c0 => color ( light red )
rule [translate-hexcode-encountered-hexcode-normal-red]:
    TranslateHexcode xff0000 => color ( normal red )
rule [translate-hexcode-encountered-hexcode-dark-red]:
    TranslateHexcode xc00000 => color ( dark red )
rule [translate-hexcode-encountered-hexcode-light-yellow]:
    TranslateHexcode xffffc0 => color ( light yellow )
rule [translate-hexcode-encountered-hexcode-normal-yellow]:
    TranslateHexcode xffff00 => color ( normal yellow )
rule [translate-hexcode-encountered-hexcode-dark-yellow]:
    TranslateHexcode xc0c000 => color ( dark yellow )
rule [translate-hexcode-encountered-hexcode-light-green]:
    TranslateHexcode xc0ffc0 => color ( light green )
rule [translate-hexcode-encountered-hexcode-normal-green]:
    TranslateHexcode x00ff00 => color ( normal green )
rule [translate-hexcode-encountered-hexcode-dark-green]:
    TranslateHexcode x00c000 => color ( dark green )
rule [translate-hexcode-encountered-hexcode-light-cyan]:
    TranslateHexcode xc0ffff => color ( light cyan )
rule [translate-hexcode-encountered-hexcode-normal-cyan]:
    TranslateHexcode x00ffff => color ( normal cyan )
rule [translate-hexcode-encountered-hexcode-dark-cyan]:
    TranslateHexcode x00c0c0 => color ( dark cyan )
rule [translate-hexcode-encountered-hexcode-light-blue]:
    TranslateHexcode xc0c0ff => color ( light blue )
rule [translate-hexcode-encountered-hexcode-normal-blue]:
    TranslateHexcode x0000ff => color ( normal blue )
```

```
rule [translate-hexcode-encountered-hexcode-dark-blue]:
    TranslateHexcode x0000c0 => color ( dark blue )
rule [translate-hexcode-encountered-hexcode-light-magenta]:
    TranslateHexcode xffc0ff => color ( light magenta )
rule [translate-hexcode-encountered-hexcode-normal-magenta]:
    TranslateHexcode xff00ff => color ( normal magenta )
rule [translate-hexcode-encountered-hexcode-dark-magenta]:
    TranslateHexcode xc000c0 => color ( dark magenta )

rule [translate-hexcode-encountered-hexcode-black]:
    TranslateHexcode x000000 => color ( black )
rule [translate-hexcode-encountered-hexcode-white]:
    TranslateHexcode xffffff => color ( white )
```

Additional colours (such as orange, brown) may be used, though their effect is implementation-dependent. In the simplest case, non-standard colours are treated by the language interpreter as the same as white, so may be used freely wherever white is used. (Another possibility is that they are treated the same as black.)

```
rule [translate-hexcode-encountered-illegal]:
    TranslateHexcode _:Id => color ( white )

configuration ⟨T⟩
    ⟨program⟩ .Map ⟨/program⟩ //maps position to colour of the pixel there

    ⟨input color="magenta" stream = "stdin"⟩ .List ⟨/input⟩
    ⟨output color="Orchid" stream = "stdout"⟩ .List ⟨/output⟩

    ⟨log⟩ .List ⟨/log⟩

    //used when building up ⟨program⟩
    ⟨buildingx⟩-1⟨/buildingx⟩
    ⟨buildingy⟩-1⟨/buildingy⟩
    ⟨nextLines⟩ . ⟨/nextLines⟩


    //used when building items in ⟨block⟩
    ⟨blockworkspace⟩ .List ⟨/blockworkspace⟩
    ⟨nextBlockID⟩ 0 ⟨/nextBlockID⟩
    ⟨path⟩ .List⟨/path⟩
```

Codels

Piet code takes the form of graphics made up of the recognised colours. Individual pixels of colour are significant in the language, so it is common for programs to be enlarged for viewing so that the details are easily visible. In such enlarged programs, the term ”codel” is used to mean a block of colour equivalent to a single pixel of code, to avoid confusion with the actual pixels of the enlarged graphic, of which many may make up one codel.

```
    ⟨k⟩ $PGM:Program ⟨/k⟩
```

Colour Blocks

The basic unit of Piet code is the colour block. A colour block is a contiguous block of any number of codels of one colour, bounded by blocks of other colours or by the edge of the program graphic. Blocks of colour adjacent only diagonally are not considered contiguous. A colour block may be any shape and may have ”holes” of other colours inside it, which are not considered part of the block.

```
⟨owner⟩ .Map ⟨/owner⟩ //maps position to the block that codel is in

    //stores all the blocks that have been constructed so far.
⟨blocks⟩
    ⟨block multiplicity = "*" type="Map"⟩
        ⟨id⟩ -1 ⟨/id⟩
        ⟨colour⟩ color(black) ⟨/colour⟩
        ⟨size⟩ -1 ⟨/size⟩
        ⟨transitions⟩ .Map ⟨/transitions⟩
    ⟨/block⟩
⟨/blocks⟩
```

Stack

Piet uses a stack for storage of all data values. Data values exist only as integers, though they may be read in or printed as Unicode character values with appropriate commands.

The stack is notionally infinitely deep, but implementations may elect to provide a finite maximum stack size. If a finite stack overflows, it should be treated as a runtime error, and handling this will be implementation dependent.

```
⟨stack⟩ .List ⟨/stack⟩
```

Program Execution

The Piet language interpreter begins executing a program in the colour block which includes the upper left codel of the program. The interpreter maintains a *Direction Pointer* (DP), initially pointing to the right. The DP may point either right, left, down or up. The interpreter also maintains a *Codel Chooser* (CC), initially pointing left. The CC may point either left or right. The directions of the DP and CC will often change during program execution.

```
⟨DP⟩DP (⟩)⟨/DP⟩
⟨CC⟩CC (⟨)⟨/CC⟩
⟨PP⟩ point(0,0) ⟨/PP⟩ //program pointer, points to current pixel
⟨exitedPP⟩ point(0,0)⟨/exitedPP⟩
⟨timesToggled⟩ 0 ⟨/timesToggled⟩

⟨/T⟩
```

As it executes the program, the interpreter traverses the colour blocks of the program under the following rules:

1. The interpreter finds the edge of the current colour block which is furthest in the direction of the DP. (This edge may be disjoint if the block is of a complex shape.)
2. The interpreter finds the codel of the current colour block on that edge which is furthest to the CC's direction of the DP's direction of travel. (Visualise this as standing on the program and walking in the direction of the DP; see table at right.)
3. The interpreter travels from that codel into the colour block containing the codel immediately in the direction of the DP.

The interpreter continues doing this until the program terminates.

DP	CC	Codel Chosen
right	left	uppermost
	right	lowermost
down	left	rightmost
	right	leftmost
left	left	lowermost
	right	uppermost
right	left	leftmost
	right	rightmost

```
syntax Direction ::= "direction" "(" DirectionPointer "," CodelChooser ")"
syntax State ::= "step" | "build" "(" Int ")" | "makeBlock" "(" Coord "," Int ")"

// Summary of a step
// We begin: PP at location, ⟨currentColour⟩ holds colour of the current block,
//    DP/CC in some configuration
// Find item in ⟨block⟩ mapped to ⟨PP⟩. Look up with DP/CC where the PP is next.
//    Update ⟨PP⟩ to this position., and ⟨exitedPP⟩ to the old ⟨PP⟩
//      If this mapping does not exist in ⟨block⟩, create the block!
// Take the colour at the new ⟨PP⟩, run TranslateInstruction on it and ⟨currentColour⟩,
// if ⟨k⟩ is step, do again

rule [next-step-mapping-exists]:
    ⟨k⟩ step => TranslateInstruction OldColour NewColour ... ⟨/k⟩
    ⟨PP⟩ OldPP:Coord => NewPP +Coord DPToOffset(D) ⟨/PP⟩
    ⟨exitedPP⟩ _ => OldPP ⟨/exitedPP⟩
    ⟨program⟩ ... (NewPP +Coord DPToOffset(D)) |-> NewColour:Colour ⋯⟨/program⟩
    ⟨owner⟩ ... OldPP |-> OldBlockID:Int ... ⟨/owner⟩
    ⟨blocks⟩
    ...
        ⟨block⟩
            ⟨id⟩ OldBlockID ⟨/id⟩
            ⟨colour⟩ OldColour:Colour ⟨/colour⟩
            ⟨size⟩ _Size:Int ⟨/size⟩
            ⟨transitions⟩ ... direction(D,C) |-> NewPP:Coord ... ⟨/transitions⟩
        ⟨/block⟩
    ...
    ⟨/blocks⟩
```

```
    ⟨DP⟩D:DirectionPointer⟨/DP⟩
    ⟨CC⟩C:CodelChooser⟨/CC⟩
    ⟨blockworkspace⟩ .List ⟨/blockworkspace⟩

rule [next-step-out-of-bounds]:
    ⟨k⟩ step => TranslateInstruction OldColour color(black) ... ⟨/k⟩
    //going out of bounds is treated like walking into a black pixel
    ⟨PP⟩ OldPP:Coord => NewPP +Coord DPToOffset(D) ⟨/PP⟩
    ⟨exitedPP⟩ _ => OldPP ⟨/exitedPP⟩
    ⟨program⟩ Program:Map ⟨/program⟩  //out of bounds, so it is unmapped
    ⟨owner⟩ ... OldPP |-> OldBlockID:Int ... ⟨/owner⟩
    ⟨blocks⟩
    ...
        ⟨block⟩
            ⟨id⟩ OldBlockID ⟨/id⟩
            ⟨colour⟩ OldColour:Colour ⟨/colour⟩
            ⟨size⟩ _Size:Int ⟨/size⟩
            ⟨transitions⟩ ... direction(D,C) |-> NewPP:Coord ... ⟨/transitions⟩
        ⟨/block⟩
    ...
    ⟨/blocks⟩
    ⟨DP⟩D:DirectionPointer⟨/DP⟩
    ⟨CC⟩C:CodelChooser⟨/CC⟩
    ⟨blockworkspace⟩ .List ⟨/blockworkspace⟩
        requires notBool ((NewPP +Coord DPToOffset(D)) in_keys (Program))

rule [next-step-no-mapping]:
    ⟨k⟩ step => makeBlock(OldPP , BlockID) ~> build(BlockID)  ~> step... ⟨/k⟩
    ⟨PP⟩ OldPP:Coord ⟨/PP⟩
    ⟨program⟩ ... OldPP |-> OldColour:Colour ⋯⟨/program⟩  //this pixel is in bounds
    ⟨owner⟩ M:Map ⟨/owner⟩
    //create a new block
    ⟨blocks⟩ ... (.Bag => ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩OldColour⟨/colour⟩
            ⟨size⟩0⟨/size⟩
            ⟨transitions⟩
                direction(DP(⟩), CC(⟩)) |-> OldPP
                direction(DP(⟩), CC(⟨)) |-> OldPP
                direction(DP(v), CC(⟩)) |-> OldPP
                direction(DP(v), CC(⟨)) |-> OldPP
                direction(DP(⟨), CC(⟩)) |-> OldPP
                direction(DP(⟨), CC(⟨)) |-> OldPP
                direction(DP(^), CC(⟩)) |-> OldPP
                direction(DP(^), CC(⟨)) |-> OldPP
            ⟨/transitions⟩
        ⟨/block⟩) ...
    ⟨/blocks⟩
    ⟨nextBlockID⟩ BlockID:Int => BlockID +Int 1 ⟨/nextBlockID⟩
    requires notBool(OldPP in_keys (M))

rule [build-block]:
    ⟨k⟩ makeBlock (point(X:Int, Y:Int ), BlockID :Int) =>
            makeBlock (point(X +Int 1, Y),BlockID) ~>
            makeBlock (point(X, Y +Int 1),BlockID) ~>
            makeBlock (point(X, Y -Int 1),BlockID) ~>
```

```
            makeBlock (point(X -Int 1, Y),BlockID)
        ⋯⟨/k⟩
    ⟨program⟩ ... point(X,Y) |-> OtherColour:Colour ... ⟨/program⟩
    // The current position is in bounds
    ⟨owner⟩ Owner:Map ⟨/owner⟩
    ⟨blocks⟩
        ...
        ⟨block⟩ ⟨id⟩BlockID⟨/id⟩ ⟨colour⟩MainColour:Colour⟨/colour⟩⋯⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ ... .List => ListItem(point(X,Y)) ⟨/blockworkspace⟩
    requires (MainColour ==Colour OtherColour)
        //This is part of the same block that we started from
        andBool notBool (point(X,Y) in_keys(Owner))
        // the current position is not already mapped
        andBool notBool(MainColour ==Colour color(white))
        //white blocks follow different rules

rule [build-block-white]:
    ⟨k⟩ makeBlock (point(X:Int, Y:Int ), BlockID :Int) => . ⋯⟨/k⟩
    ⟨program⟩ ... point(X,Y) |-> OtherColour:Colour ... ⟨/program⟩
    // The current position is in bounds
    ⟨owner⟩ Owner:Map ⟨/owner⟩
    ⟨blocks⟩
        ...
        ⟨block⟩ ⟨id⟩BlockID⟨/id⟩ ⟨colour⟩MainColour:Colour⟨/colour⟩⋯⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ ... .List => ListItem(point(X,Y)) ⟨/blockworkspace⟩
    requires (MainColour ==Colour color(white) )
        //white pixels make single-pixel blocks
        andBool notBool (point(X,Y) in_keys(Owner))
        // the current position is not already mapped


//do not add the pixel into the current block, for some reason;
//i.e out of bounds, wrong colour, already in block
rule [build-block-wrong-pixel-wrong-colour]:
    ⟨k⟩ makeBlock (Position:Coord, BlockID:Int) => . ⋯⟨/k⟩
    ⟨program⟩ ... Position |-> OtherColour:Colour ... ⟨/program⟩
    ⟨blocks⟩
        ...
        ⟨block⟩ ⟨id⟩BlockID⟨/id⟩ ⟨colour⟩MainColour:Colour⟨/colour⟩⋯⟨/block⟩
        ...
    ⟨/blocks⟩
    requires notBool (MainColour ==Colour OtherColour)
    //This is not part of the block that we started from



rule [build-block-wrong-pixel-out-bounds]:
    ⟨k⟩ makeBlock (Position:Coord, _BlockID:Int) => . ⋯⟨/k⟩
    ⟨program⟩ Program:Map ⟨/program⟩
    ⟨owner⟩ Owner:Map ⟨/owner⟩
    ⟨blockworkspace⟩ B:List⟨/blockworkspace⟩
    requires Position in_keys(Owner) // the current position is already mapped
```

```
        orBool notBool (Position in_keys(Program))// The current position is out of bounds
        orBool Position in(B) // the current position is already mapped

//the following rules are all very similar and do the same thing
//Add the pixel to a block as the most extreme pixel in some direction
//⟨blockworkspace⟩ contains all the pixels in the current block we are building.
//So we should deplete thet cell until it is empty, to create the block
rule [build-block-rightright]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩_ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(⟩), CC(⟩)) |—> (point(X2:Int,Y2:Int) => point(X1,Y1))
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (X1 >Int X2)
        orBool (X1 ==Int X2 andBool Y1 >Int Y2)
        //The new point is the new right-bottom pixel in the block

rule [build-block-rightleft]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(⟩), CC(⟨)) |—> (point(X2:Int,Y2:Int) =>  point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (X1 >Int X2)
        orBool (X1 ==Int X2 andBool  Y2 >Int Y1 )
        //The new point is the new right-top pixel in the block

rule [build-block-downright]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
```

```
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(v), CC(⟩)) |−> (point(X2:Int,Y2:Int) => point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (Y1 >Int Y2)
        orBool (Y1 ==Int Y2 andBool X2 >Int X1 )
        //The new point is the new bottom-left pixel in the block

rule [build-block-downleft]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(v), CC(⟨)) |−> (point(X2:Int,Y2:Int)  => point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (Y1 >Int Y2)
        orBool (Y1 ==Int Y2 andBool X1 >Int X2 )
        //The new point is the new bottom-right pixel in the block

rule [build-block-leftleft]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(⟨), CC(⟨)) |−> (point(X2:Int,Y2:Int) => point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (X2 >Int X1)
        orBool (X1 ==Int X2 andBool Y1 >Int Y2 )
        //The new point is the new left-bottom pixel in the block
```

```
rule [build-block-leftright]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(⟨), CC(⟩)) |-> (point(X2:Int,Y2:Int) => point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (X2 >Int X1)
        orBool (X1 ==Int X2 andBool Y2 >Int Y1 )
        //The new point is the new left-top pixel in the block

rule [build-block-upleft]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(^), CC(⟨)) |-> (point(X2:Int,Y2:Int) =>  point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (Y2 >Int Y1)
        orBool (Y1 ==Int Y2 andBool X2 >Int X1 )
        //The new point is the new top-left pixel in the block

rule [build-block-upright]:
    ⟨k⟩ build (BlockID:Int)  …⟨/k⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩ _ ⟨/colour⟩
            ⟨size⟩ _ ⟨/size⟩
            ⟨transitions⟩
                ...
                direction(DP(^), CC(⟩)) |-> (point(X2:Int,Y2:Int)  =>  point(X1,Y1) )
                ...
            ⟨/transitions⟩
        ⟨/block⟩
        ...
```

```
    ⟨/blocks⟩
    ⟨blockworkspace⟩ListItem(point(X1:Int,Y1:Int)) ... ⟨/blockworkspace⟩
    requires (Y2 >Int Y1)
        orBool (Y1 ==Int Y2 andBool X1 >Int X2 )
        //The new point is the new top-right pixel in the block


//this pixel does not /no longer updates the block edges,
//so just add it to the block mapping,and increment block size
rule [build-block-finish-up]:
    ⟨k⟩ build (BlockID:Int) …⟨/k⟩
    ⟨owner⟩ ... .Map => point(X,Y) |-> BlockID⟨/owner⟩
    ⟨blocks⟩
        ...
        ⟨block⟩
            ⟨id⟩BlockID⟨/id⟩
            ⟨colour⟩_⟨/colour⟩
            ⟨size⟩Size:Int => Size+Int 1⟨/size⟩
            ⟨transitions⟩
                direction(DP(⟩), CC(⟩)) |-> point(RRX:Int, RRY:Int)
                direction(DP(⟩), CC(⟨)) |-> point(RLX:Int, RLY:Int)
                direction(DP(v), CC(⟩)) |-> point(DRX:Int, DRY:Int)
                direction(DP(v), CC(⟨)) |-> point(DLX:Int, DLY:Int)
                direction(DP(⟨), CC(⟩)) |-> point(LRX:Int, LRY:Int)
                direction(DP(⟨), CC(⟨)) |-> point(LLX:Int, LLY:Int)
                direction(DP(^), CC(⟩)) |-> point(URX:Int, URY:Int)
                direction(DP(^), CC(⟨)) |-> point(ULX:Int, ULY:Int)
            ⟨/transitions⟩
        ⟨/block⟩
        ...
    ⟨/blocks⟩
    ⟨blockworkspace⟩ ListItem(point(X:Int,Y:Int)) => .List ... ⟨/blockworkspace⟩
    requires    notBool ((X >Int RRX)  orBool (X ==Int RRX andBool Y >Int RRY))
            //not the right-bottom
        andBool notBool ((X >Int RLX)  orBool (X ==Int RLX andBool RLY >Int Y))
            //not the right-top
        andBool notBool ((Y >Int DRY)  orBool (Y ==Int DRY andBool DRX >Int X))
            //not the bottom-left
        andBool notBool ((Y >Int DLY)  orBool (Y ==Int DLY andBool X >Int DLX))
            //not the bottom-right
        andBool notBool ((LRX >Int X)  orBool (X ==Int LRX andBool LRY >Int Y))
            //not the left-top
        andBool notBool ((LLX >Int X)  orBool (X ==Int LLX andBool Y >Int LLY))
            //not the left-bottom
        andBool notBool ((URY >Int Y)  orBool (Y ==Int URY andBool X >Int URX))
            //not the top-right
        andBool notBool ((ULY >Int Y)  orBool (Y ==Int ULY andBool ULX >Int X))
            //not the top-left

rule [build-block-finished]: //we have finished constructing the block
    ⟨k⟩ build(_) => . …⟨/k⟩
    ⟨blockworkspace⟩.List⟨/blockworkspace⟩
```

Syntax Elements

Numbers

Each non-black, non-white colour block in a Piet program represents an integer equal to the number of codels in that block. Note that non-positive integers cannot be represented, although they can be constructed with operators. When the interpreter encounters a number, it does not necessarily do anything with it. In particular, it is not automatically pushed on to the stack - there is an explicit command for that (see below).

The maximum size of integers is notionally infinite, though implementations may implement a finite maximum integer size. An integer overflow is a runtime error, and handling this will be implementation dependent.

Black Blocks and Edges

Black colour blocks and the edges of the program restrict program flow. If the Piet interpreter attempts to move into a black block or off an edge, it is stopped and the CC is toggled. The interpreter then attempts to move from its current block again. If it fails a second time, the DP is moved clockwise one step. These attempts are repeated, with the CC and DP being changed between alternate attempts. If after eight attempts the interpreter cannot leave its current colour block, there is no way out and the program terminates.

```
rule [translate-instruction-to-black]:
    ⟨k⟩TranslateInstruction _ color(black)    => blk(TT) ⋯⟨/k⟩
    ⟨timesToggled⟩ TT:Int => TT +Int 1 ⟨/timesToggled⟩

rule [hit-black-pixel-dp]:
    ⟨k⟩blk(I:Int) => rotdp(1) ~> step ⋯⟨/k⟩
    ⟨PP⟩ _ => ExitedPP ⟨/PP⟩ //roll back the program pointer
    ⟨exitedPP⟩ ExitedPP:Coord ⟨/exitedPP⟩
    ⟨log⟩ ... .List => ListItem("bounced dp") ⟨/log⟩
    requires notBool (I ==Int 7) andBool (I %Int 2 ==Int 1)
rule [hit-black-pixel-c]:
    ⟨k⟩blk(I:Int) => rotcc(1) ~> step ⋯⟨/k⟩
    ⟨PP⟩ _ => ExitedPP ⟨/PP⟩ //roll back the program pointer
    ⟨exitedPP⟩ ExitedPP:Coord ⟨/exitedPP⟩
    ⟨log⟩ ... .List => ListItem("bounced cc") ⟨/log⟩
    requires notBool (I ==Int 7) andBool (I %Int 2 ==Int 0)
rule [hit-black-pixel-too-many]:
    ⟨k⟩blk(7) => stop ⋯⟨/k⟩
    ⟨log⟩ ... .List => ListItem("stuck") ⟨/log⟩
```

White Blocks

White colour blocks are ”free” zones through which the interpreter passes unhindered. If it moves from a colour block into a white area, the interpreter ”slides” through the white codels in the direction of the DP until it reaches a non-white colour block. If the interpreter slides into a black block or an edge, it is considered restricted (see above), otherwise it moves into the colour block so encountered. Sliding across white blocks

into a new colour does not cause a command to be executed (see below). In this way, white blocks can be used to change the current colour without executing a command, which is very useful for coding loops.

Sliding across white blocks takes the interpreter in a *straight line* until it hits a coloured pixel or edge. It does not use the procedure described above for determining where the interpreter emerges from non-white coloured blocks.

Precisely what happens when the interpeter slides across a white block and hits a black block or an edge was not clear in the original specification. My interpretation follows from a literal reading of the above text:

- The interpreter ”slides” across the white block in a straight line.
- If it hits a restriction, the CC is toggled. Since this results in no difference in where the interpreter is trying to go, the DP is immediately stepped clockwise.
- The interpreter now begins sliding from its current white codel, in the new direction of the DP, until it either enters a coloured block or encounters another restriction.
- Each time the interpreter hits a restriction while within the white block, it toggles the CC and steps the DP clockwise, then tries to slide again. This process repeats until the interpreter either enters a coloured block (where execution then continues); or until the interpreter begins retracing its route. If it retraces its route entirely within a white block, there is no way out of the white block and execution should terminate.

```
rule [translate-instruction-from-white]:
    TranslateInstruction color(white) color(_ _)   => nop
rule [translate-instruction-to-white]:
    TranslateInstruction color(_ _) color(white)    => nop
rule [translate-instruction-between-white]:
    TranslateInstruction color(white) color(white)    => nopW
```

Commands

Commands are defined by the transition of colour from one colour block to the next as the interpreter travels through the program. The number of steps along the Hue Cycle and Lightness Cycle in each transition determine the command executed, as shown in the table at right. If the transition between colour blocks occurs via a slide across a white block, no command is executed. The individual commands are explained below.

```
rule [translate-instruction-colours]:
    TranslateInstruction color(L1 H1) color(L2 H2)
     => LookupInstruction LightnessDifference L1 L2 HueDifference H1 H2
```

- **push:** Pushes the value of the colour block just exited on to the stack. Note that values of colour blocks are not automatically pushed on to the stack - this push operation must be explicitly carried out.

```
rule [instruction-push]:
    ⟨k⟩ push => nop ⟨/k⟩
    ⟨exitedPP⟩PP:Coord ⟨/exitedPP⟩
    ⟨owner⟩ ... PP |-> ID:Int ⋯⟨/owner⟩
    ⟨blocks⟩
    ...
        ⟨block⟩
            ⟨id⟩ ID ⟨/id⟩
            ⟨colour⟩_⟨/colour⟩
            ⟨size⟩ I:Int⟨/size⟩
            ⟨transitions⟩_⟨/transitions⟩
        ⟨/block⟩
    ...
    ⟨/blocks⟩
    ⟨stack⟩ .List => ListItem(I) ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("PUSH,") ListItem(I)⟨/log⟩
```

- **pop:** Pops the top value off the stack and discards it.

```
rule [instruction-pop]:
    ⟨k⟩ pop => nop⟨/k⟩
    ⟨stack⟩ ListItem(_) => .List ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("POP,") ⟨/log⟩
```

- **add:** Pops the top two values off the stack, adds them, and pushes the result back on the stack.

```
rule [instruction-add]:
    ⟨k⟩ add => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int) ListItem(I2:Int)
        => ListItem(I1 +Int I2)
        ...
        ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("ADD,") ⟨/log⟩
```

- **subtract:** Pops the top two values off the stack, calculates the second top value minus the top value, and pushes the result back on the stack.

```
rule [instruction-subtract]:
    ⟨k⟩ sub => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int) ListItem(I2:Int)
        => ListItem(I2 -Int I1)
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("SUB,") ⟨/log⟩
```

- **multiply:** Pops the top two values off the stack, multiplies them, and pushes the result back on the stack.

```
rule [instruction-multiply]:
    ⟨k⟩ mult => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int) ListItem(I2:Int)
        => ListItem(I2 *Int I1)
```

```
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("MU<,") ⟨/log⟩
```

- **divide:** Pops the top two values off the stack, calculates the integer division of the second top value by the top value, and pushes the result back on the stack. If a divide by zero occurs, it is handled as an implementation-dependent error, though simply ignoring the command is recommended.

```
rule [instruction-divide]:
    ⟨k⟩ div => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int) ListItem(I2:Int)
        => ListItem(I2 /Int I1)
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("DIV,") ⟨/log⟩
        requires notBool (I1 ==Int 0)

rule [instruction-divide-by-zero]:
    ⟨k⟩ div => nop⟨/k⟩
    ⟨stack⟩ ListItem(0) ListItem(_:Int) ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("DIV,") ⟨/log⟩
```

- **mod:** Pops the top two values off the stack, calculates the second top value modulo the top value, and pushes the result back on the stack. The result has the same sign as the divisor (the top value). If the top value is zero, this is a divide by zero error, which is handled as an implementation-dependent error, though simply ignoring the command is recommended. (*See note below.*)

```
//note: This assumes that the inbuilt %Int is modulus that has same sign as
//the *dividend* rather than divisor, there's the possibility I've
//misunderstood the docs
rule [instruction-modulo-positive]:
    //divisor is positive, so output must be positive
    ⟨k⟩ mod => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int) ListItem(I2:Int)
        => ListItem(I2 %Int I1)
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("MOD,") ⟨/log⟩
        requires notBool (I1 ==Int 0) andBool (I1 >Int 0)
rule [instruction-modulo-negative]:
    //divisor is negative, so so it output
    ⟨k⟩ mod => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int) ListItem(I2:Int)
        => ListItem(0 -Int ((0 -Int I2) %Int (0 -Int I1)))
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("MOD,") ⟨/log⟩
        requires notBool (I1 ==Int 0) andBool (0 >Int I1)

rule [instruction-modulo-zero]: // ignoring a mod by zero
```

```
    ⟨k⟩ mod => nop⟨/k⟩
    ⟨stack⟩ ListItem(0) ListItem(_Int)... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("MOD,") ⟨/log⟩
```

- **not:** Replaces the top value of the stack with 0 if it is non-zero, and 1 if it is zero.

```
rule [instruction-not-nonzero]:
    ⟨k⟩ not => nop⟨/k⟩
    ⟨stack⟩
        ListItem(I1:Int)
        => #if I1 ==Int 0 #then ListItem(1) #else ListItem(0) #fi
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("NOT,") ⟨/log⟩
```

- **greater:** Pops the top two values off the stack, and pushes 1 on to the stack if the second top value is greater than the top value, and pushes 0 if it is not greater.

```
rule [instruction-greater]:
    ⟨k⟩ great => nop⟨/k⟩
    ⟨stack⟩
        ListItem(Top:Int) ListItem(Bottom:Int)
        => #if Bottom >Int Top #then ListItem(1) #else ListItem(0) #fi
        ...
    ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("GR") ⟨/log⟩
```

- **pointer:** Pops the top value off the stack and rotates the DP clockwise that many steps (anticlockwise if negative).

```
rule [instruction-pointer]:
    ⟨k⟩ ptr => rotdp(abs(Steps)) ~> nop ⋯⟨/k⟩
    ⟨stack⟩ListItem(Steps:Int) => .List ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("PTR") ⟨/log⟩


syntax State ::= "rotdp" "(" Int ")" [strict]

rule   ⟨k⟩rotdp(0) => . ⋯⟨/k⟩
rule   ⟨k⟩rotdp(X:Int) => rotdp(X -Int 1) ⋯⟨/k⟩
       ⟨DP⟩DP(⟩) => DP(v)⟨/DP⟩
       requires notBool (X ==Int 0)
rule   ⟨k⟩rotdp(X:Int) => rotdp(X -Int 1) ⋯⟨/k⟩
       ⟨DP⟩DP(v) => DP(⟨)⟨/DP⟩
       requires notBool (X ==Int 0)
rule   ⟨k⟩rotdp(X:Int) => rotdp(X -Int 1) ⋯⟨/k⟩
       ⟨DP⟩DP(⟨) => DP(^)⟨/DP⟩
       requires notBool (X ==Int 0)
rule   ⟨k⟩rotdp(X:Int) => rotdp(X -Int 1) ⋯⟨/k⟩
       ⟨DP⟩DP(^) => DP(⟩)⟨/DP⟩
       requires notBool (X ==Int 0)
```

- **switch:** Pops the top value off the stack and toggles the CC that many times (the absolute value of that many times if negative).

```
rule [instruction-switch]:
```

```
    ⟨k⟩ switch => rotcc(abs(C)) ~> nop ⋯⟨/k⟩
    ⟨stack⟩ListItem(C:Int) => .List ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("SWITCH") ⟨/log⟩


syntax State ::= "rotcc" "(" Int ")" [strict]
rule   ⟨k⟩rotcc(0) => . ⋯⟨/k⟩
rule   ⟨k⟩rotcc(X:Int) => rotcc(X -Int 1) ⋯⟨/k⟩
       ⟨CC⟩CC(⟩) => CC(⟨)⟨/CC⟩
       requires notBool (X ==Int 0)
rule   ⟨k⟩rotcc(X:Int) => rotcc(X -Int 1) ⋯⟨/k⟩
       ⟨CC⟩CC(⟨) => CC(⟩)⟨/CC⟩
       requires notBool (X ==Int 0)
```

- **duplicate:** Pushes a copy of the top value on the stack on to the stack.

```
rule [instruction-duplicate]:
    ⟨k⟩ dup => nop⟨/k⟩
    ⟨stack⟩ListItem(Value:Int) => ListItem(Value) ListItem(Value) ⋯⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("DUP") ⟨/log⟩
```

- **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.

```
rule [instruction-roll]:
    ⟨k⟩roll => rollby(Depth, NumRolls)⟨/k⟩
    ⟨stack⟩ ListItem(NumRolls:Int) ListItem(Depth:Int) => .List ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("ROLL,") ⟨/log⟩
        requires Depth ⟩=Int 0

rule [instruction-roll-negative-depth]:
    ⟨k⟩roll => nop⟨/k⟩
    ⟨stack⟩ ListItem(_:Int) ListItem(Depth:Int) => .List ... ⟨/stack⟩
        requires 0 >Int Depth //negative depth is an error and ignored


syntax State ::= "rollby" "(" Int "," Int ")" [strict]
rule   ⟨k⟩rollby(_:Int, 0) => nop⟨/k⟩

rule   ⟨k⟩ rollby (Depth:Int, _:Int) => nop ... ⟨/k⟩ //ignore if depth too large
       ⟨stack⟩ S:List ⟨/stack⟩
           requires (Depth >Int size(S))

rule   ⟨k⟩rollby(Depth:Int, NumRolls:Int) => rollby(Depth, NumRolls -Int 1)⟨/k⟩
       ⟨stack⟩
           S:List =>
           (range(S, 1, size(S) -Int Depth )
           range(S, 0, size(S) -Int 1)
```

```
            range(S, Depth,0) )
        ⟨/stack⟩
            requires NumRolls >Int 0

// other (presumably correct) implementations take the value at [depth]
//and move it to the top of the stack, so that's what I'll do.
// I originall thought it meant taking the value at the top and moving
//it to [depth] from the *bottom* of the stack
rule    ⟨k⟩rollby(Depth:Int, NumRolls:Int) => rollby(Depth, NumRolls +Int 1)⟨/k⟩
        ⟨stack⟩
            S:List =>
            (range(S, Depth, size(S) -Int Depth -Int 1  )
            range(S, 0, size(S) -Int Depth )
            range(S, Depth +Int 1,0) )
        ⟨/stack⟩
            requires 0 >Int NumRolls
```

- **in:** Reads a value from STDIN as either a number or character, depending on the particular incarnation of this command and pushes it on to the stack. If no input is waiting on STDIN, this is an error and the command is ignored. If an integer read does not receive an integer value, this is an error and the command is ignored.

```
rule [instruction-in-number-int]:
    ⟨k⟩innum => nop ⟨/k⟩
    ⟨stack⟩ .List => ListItem(I) ... ⟨/stack⟩
    ⟨log⟩ ... .List => ListItem ("INNUM,") ⟨/log⟩
    ⟨input⟩ ListItem(I:Int) => .List ⋯⟨/input⟩

rule [instruction-in-char]:
    ⟨k⟩inchar => nop⟨/k⟩
    ⟨stack⟩ .List => ListItem(#getc(#stdin)) ... ⟨/stack⟩
    //yes, we're using FFI, I'm very sorry for this, I managed to avoid it until now.
    ⟨log⟩ ... .List => ListItem ("INCHAR,") ⟨/log⟩
```

- **out:** Pops the top value off the stack and prints it to STDOUT as either a number or character, depending on the particular incarnation of this command.

```
rule [instruction-out-number]:
    ⟨k⟩ outnum => nop⟨/k⟩
    ⟨stack⟩ListItem(Value:Int) => .List ⋯⟨/stack⟩
    ⟨output⟩ ... .List => ListItem(Value)⟨/output⟩
    ⟨log⟩ ... .List => ListItem ("OUTNUM") ⟨/log⟩

rule [instruction-out-character]:
    ⟨k⟩ outchar => nop⟨/k⟩
    ⟨stack⟩ListItem(Value:Int) => .List ⋯⟨/stack⟩
    ⟨output⟩ ... .List => ListItem(chrChar(Value))⟨/output⟩
    ⟨log⟩ ... .List => ListItem ("OUTCHAR") ⟨/log⟩
```

	Lightness Change		
Hue Change	None	1 Darker	2 Darker
None		push	pop
1 Step	add	subtract	multiply
2 Step	divide	mod	not
3 Step	greater	pointer	switch
4 Step	duplicate	roll	in(number)
5 Step	in(char)	out(number)	out(char)

```
rule [instruction-resolution-none]:     LookupInstruction 0 0 => nop
rule [instruction-resolution-push]:     LookupInstruction 1 0 => push
rule [instruction-resolution-pop]:      LookupInstruction 2 0 => pop
rule [instruction-resolution-add]:      LookupInstruction 0 1 => add
rule [instruction-resolution-subtract]: LookupInstruction 1 1 => sub
rule [instruction-resolution-multiply]: LookupInstruction 2 1 => mult
rule [instruction-resolution-divide]:   LookupInstruction 0 2 => div
rule [instruction-resolution-modulo]:   LookupInstruction 1 2 => mod
rule [instruction-resolution-not]:      LookupInstruction 2 2 => not
rule [instruction-resolution-greater]:  LookupInstruction 0 3 => great
rule [instruction-resolution-pointer]:  LookupInstruction 1 3 => ptr
rule [instruction-resolution-switch]:   LookupInstruction 2 3 => switch
rule [instruction-resolution-duplicate]:LookupInstruction 0 4 => dup
rule [instruction-resolution-roll]:     LookupInstruction 1 4 => roll
rule [instruction-resolution-in(num)]:  LookupInstruction 2 4 => innum
rule [instruction-resolution-in(char)]: LookupInstruction 0 5 => inchar
rule [instruction-resolution-out(num)]: LookupInstruction 1 5 => outnum
rule [instruction-resolution-out(char)]:LookupInstruction 2 5 => outchar

//a NOP occurs after every instruction.
//Thus, we can reset the timesToggled counter, becuase we did not hit a black pixel
rule [process-nop]:
    ⟨k⟩ nop => step ⋯⟨/k⟩
    ⟨PP⟩ P:Coord ⟨/PP⟩
    ⟨program⟩ ... P |-> C:Colour ... ⟨/program⟩
    ⟨path⟩ ... .List => ListItem (WP(P,C)) ⟨/path⟩
    ⟨timesToggled⟩ _ => 0 ⟨/timesToggled⟩ //[structural]

//check that we are not in a white loop. If we are, halt the program
rule [process-nop-loop]:
    ⟨k⟩ nopW => #if repeats(L , 1) #then stop #else nop #fi ⟨/k⟩
    ⟨path⟩ L ⟨/path⟩
    ⟨log⟩ ... .List => ListItem("NOPW") ⟨/log⟩

syntax Bool ::= "repeats" "(" List "," Int ")" [strict, function]

rule repeats(L:List , I:Int) => false requires I *Int 2 >=Int size(L)


rule repeats (L:List, I:Int) => (
    (   getLastN(L, I) getLastN(L, I)) ==List getLastN(L, I*Int 2)
        andBool (notBool (hasNonWhite ( getLastN(L, I*Int 2)))))
    orBool repeats(L, I +Int 1)
    requires size(L) >Int I *Int 2
```

```
syntax Bool ::= "hasNonWhite" "("  List ")" [strict, function]

rule hasNonWhite (ListItem(WP(_:Coord, color(_:Lightness  _:Hue) )) _:List ) => true
rule hasNonWhite ((ListItem(WP(_:Coord, color(white))) L:List) ) =>  hasNonWhite (L)
rule hasNonWhite ( .List ) => false
```

Any operations which cannot be performed (such as popping values when not enough are on the stack) are simply ignored, and processing continues with the next command.

```
rule [ignore-one-arg-instruction]:
                      ⟨k⟩ _:Instruction1Arg => nop ⟨/k⟩
                      ⟨stack⟩ S:List ⟨/stack⟩
                      ⟨log⟩ ... .List => ListItem ("IGNORE,") ⟨/log⟩
                      requires 1 >Int size(S)

rule [ignore-two-arg-instruction]:
                      ⟨k⟩ _:Instruction2Arg => nop ⟨/k⟩
                      ⟨stack⟩ S:List ⟨/stack⟩
                      ⟨log⟩ ... .List => ListItem ("IGNORE,") ⟨/log⟩
                      requires 2 >Int size(S)

//####
// RELATED TO PARSING PROGRAM INTO ⟨program⟩ CELL
//####

rule [finished-parsing]:
    ⟨k⟩.Lines => step ⟨/k⟩

rule [parse-next-line]:
    ⟨k⟩L:Line ; Ls:Lines => L ⟨/k⟩
    ⟨nextLines⟩ . => Ls ⟨/nextLines⟩
    ⟨buildingx⟩ _X:Int => -1 ⟨/buildingx⟩
    ⟨buildingy⟩ Y:Int => Y +Int 1 ⟨/buildingy⟩  [structural]

rule [parse-next-line-restore]:
    ⟨k⟩ .Line => Ls ⟨/k⟩
    ⟨nextLines⟩ Ls:Lines => .⟨/nextLines⟩  [structural]

rule [parse-next-pixel]:
    ⟨k⟩P:Pixel  L:Line => P ~> L⟨/k⟩
    ⟨buildingx⟩ X:Int => X +Int 1 ⟨/buildingx⟩ [structural]

//place the colour index into the program cell, mapped from its position.
//This means we will be able to look up colours from positions later
rule [place-pixel-in-map]:
    ⟨k⟩ C:Colour => . ⋯⟨/k⟩
    ⟨buildingx⟩ X:Int ⟨/buildingx⟩
    ⟨buildingy⟩ Y:Int ⟨/buildingy⟩
    ⟨program⟩ ... .Map => point(X,Y) |-> C ⋯⟨/program⟩ [structural]

endmodule

module KPIET-SYNTAX
    imports DOMAINS
```

```
syntax Colour ::=  "color" "(" Lightness Hue ")"
                | "color" "(" "black" ")"
                | "color" "(" "white" ")"
syntax Hue ::= "red" | "yellow" | "green" | "cyan" | "blue" | "magenta"
syntax Lightness ::= "light" | "normal" | "dark"

syntax KResult ::= Int | Colour | Coord | Instruction | Bool | Waypoint

syntax Coord ::= "point" "(" Int "," Int ")"

syntax Program ::= Lines

syntax Lines ::= List{Line, ";"}
syntax Line ::= List{Pixel, ""}

syntax Pixel ::= Colour | Hexcode


syntax DirectionPointer ::= "DP" "(" "^" ")"
                        |   "DP" "(" "v" ")"
                        |   "DP" "(" "⟨" ")"
                        |   "DP" "(" "⟩" ")"
syntax CodelChooser ::= "CC" "(" "⟨"  ")"|"CC" "(" "⟩" ")"


syntax Instruction ::=  VMInstruction
                  | "TranslateInstruction" Colour Colour [strict]
                  | "LookupInstruction" Int Int [strict, function]
                  | "nop" | "nopW"
                  | "stop"
                  | "blk" "(" Int ")"


syntax VMInstruction ::=  Instruction1Arg | Instruction2Arg
                       | "push" | "innum" | "inchar"

syntax Instruction1Arg ::=   "pop"   |
                             "not"   |
                             "ptr"   |
                             "switch"|
                             "dup"   |
                             "outnum"|
                             "outchar"


syntax Instruction2Arg ::= "add"   |   "sub"   |   "mult"  |
               "div"   |   "mod"   |  "great" | "roll"


syntax Int ::=      "LightnessDifference" Lightness Lightness [function, functional]
               |   "HueDifference" Hue Hue [function, functional]


syntax Hexcode ::=  "xffc0c0"   | "xff0000" | "xc00000" |
                   "xffffc0"   | "xffff00" | "xc0c000" |
                   "xc0ffc0"   | "x00ff00" | "x00c000" |
```

```
                        "xc0ffff"   | "x00ffff" | "x00c0c0" |
                        "xc0c0ff"   | "x0000ff" | "x0000c0" |
                        "xffc0ff"   | "xff00ff" | "xc000c0" |
                        "x000000"   | "xffffff" | Id

    syntax Waypoint ::= "WP" "(" Coord "," Colour ")"
endmodule
```

Note on the mod command: In the original specification of Piet the result of a modulo operation with a negative dividend (the second top value popped off the stack) was not explicitly defined. I assumed that everyone would assume that the result of (p mod q) would always be equal to (($p + Nq$) mod q) for any integer N. So:

- 5 mod 3 = 2
- 2 mod 3 = 2
- -1 mod 3 = 2
- -4 mod 3 = 2

The mod command is thus identical to *floored division* in Wikipedia's page on the modulus operation.

Sample Programs and Resources

- Sample programs
- Third-party Piet interpreters and development tools

Home | Esoteric Programming Languages
Last updated: Thursday, 27 September, 2018; 04:00:52 PDT.

Hosted by: DreamHost

Adapted from David Morgan-Mar's original specification