

1 Introduction

Our language is essentially a friendship bracelet pattern generator. It will allow users to design their own patterns and test out whether they would actually be feasible to create beforehand (ex. no gaps in knots, strings end up in right position, etc.). Making friendship bracelets is also something a lot of people do, so we feel like a pattern generator would be a useful tool to have.

We think this would make a worthwhile programming language because if designed correctly, it will allow the generation of (possibly) infinite designs, fostering creative expression through the aid of technology. Also, we hope to make the language relatively simple and user-friendly (and perhaps with some sort of graphical interface) so that anyone, regardless of coding experience, can design with it :).

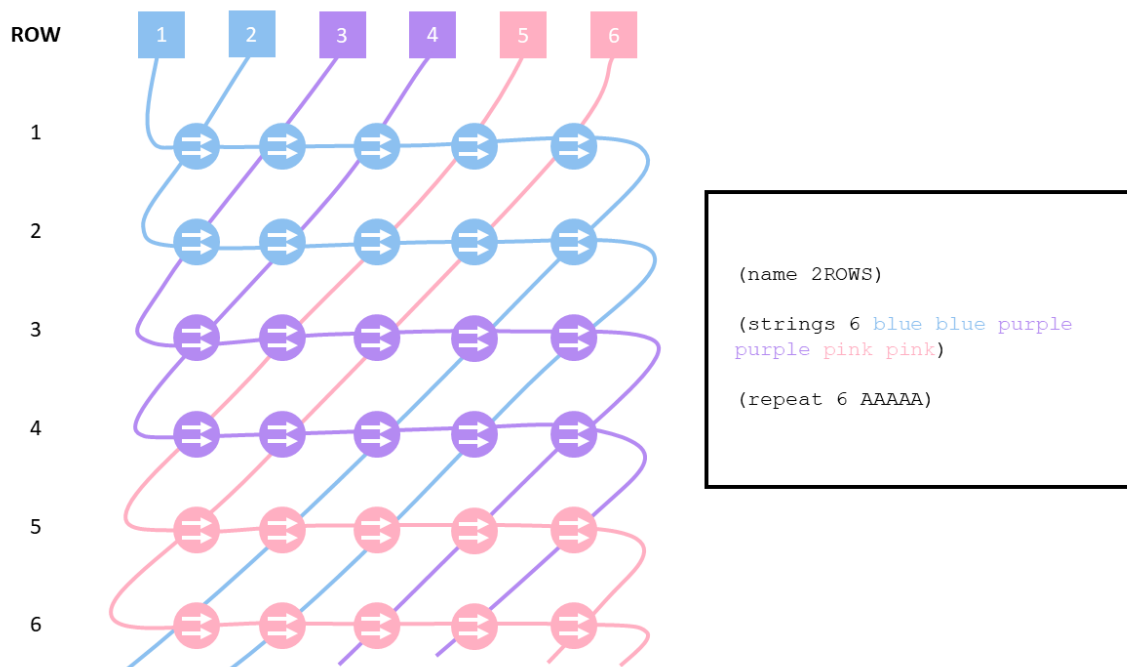
2 Design Principles

This language would definitely be guided by simplicity and clarity to keep the non-technical user in mind. We would also want the language to translate naturally from the physical art of bracelet making (ex. for a double right knot, maybe the user would type DR or something). The language would be flexible enough to allow the user's creativity to shine, while remaining relatively straight forward and easy to understand.

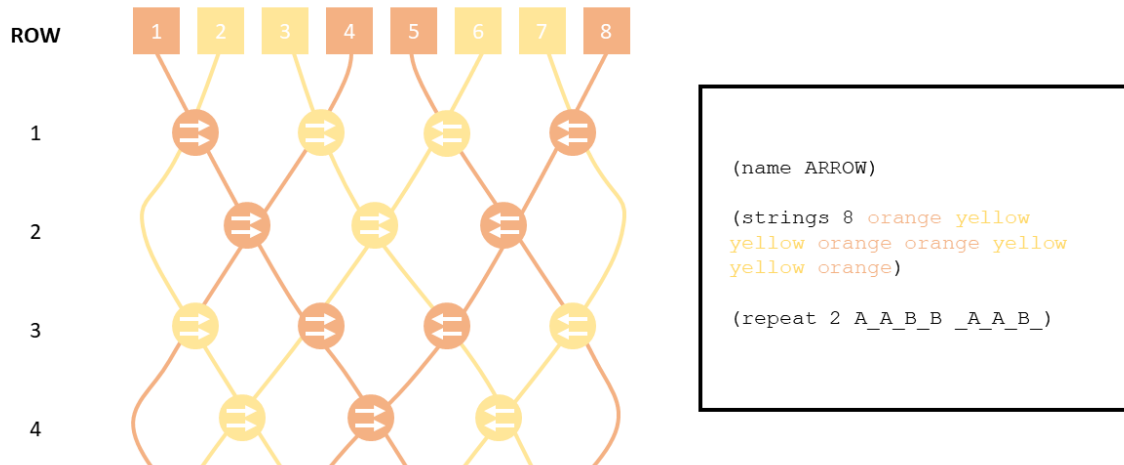
If possible, it would be cool to create an interface that simulates a digital bracelet with the user's finished pattern at the end too, so they can really visualize what it would look like. We definitely also want to provide support for users in the cases of invalid expressions, syntax errors, or other things like that.

3 Example Programs

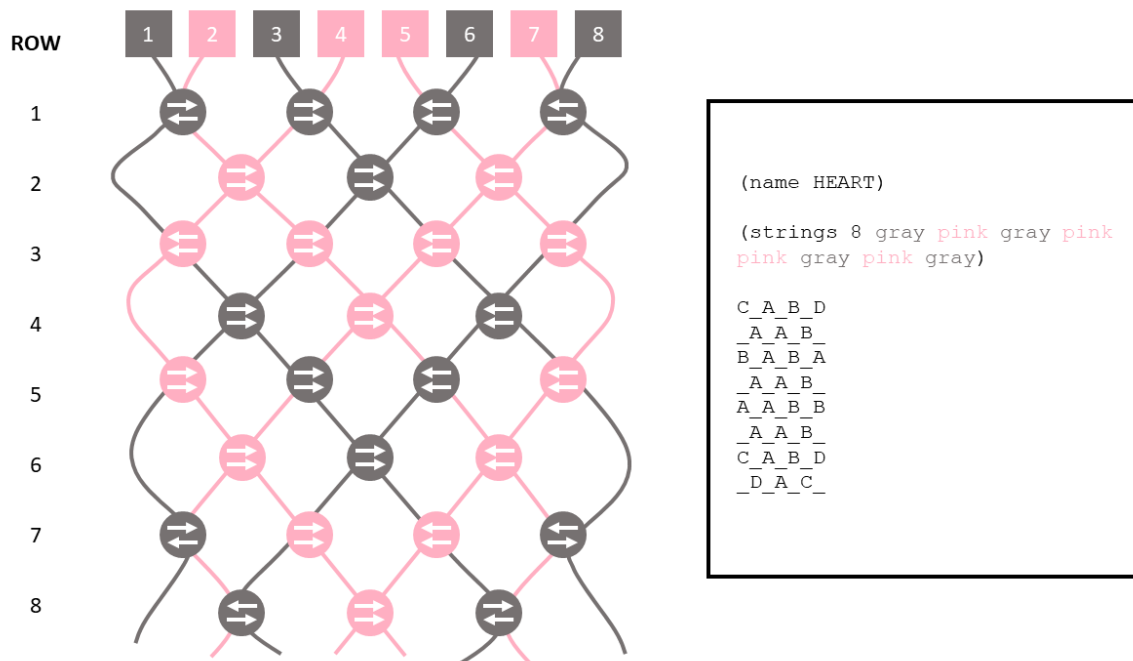
Example 1: Two row pattern



Example 2: Arrow pattern



Example 3: Heart pattern



4 Language Concepts

To write programs in this language, a user should understand the basic properties of a friendship bracelet. Essentially, a bracelet is made up of a repeated pattern, which is made up of rows, each of which consists of a certain combination of knots. There are 4 main knots: the double right knot, double left knot, right-left knot, and left-right knot. Rows can be repeated to form the pattern, or they can all be unique combinations of knots.

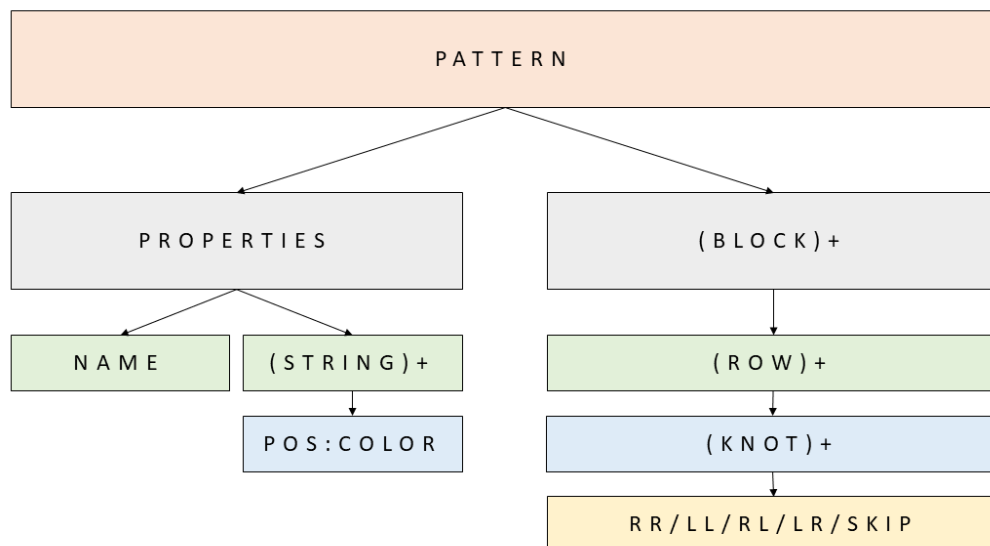
Each bracelet also has a specified number of strings, each of which can be any color. The strings have an original order to start the pattern, but usually will move around as rows are constructed (based on which knots are used and in what order). See example programs above for a basic visualization of these concepts.

5 Syntax

The syntax of this language is essentially as follows:

- The overall program creates a **pattern**. A pattern is made up of:
 - **Properties** which consist of
 - * The pattern's **name**
 - * 1+ **strings**, each having a specified color and start position.
 - 1+ **Blocks** of 1+ rows. Blocks may or may not be repeated. In examples 1 and 2, there is 1 repeated block. In example 3, there is 1 block but it is not repeated.
 - * Each **row** consists of 1+ **knots**, which are specified by one of the 4 valid knots (see the *Language Concepts* section above for more details). To simplify things, the user will use the characters "A" "B" "C" and "D" to denote the knots, and "." to skip a string (aka. basically an "empty knot").

A diagram summarizing the syntax is given below.



Here is the equivalent, formal definition of the grammar in Backus-Naur form:

```

<pattern> ::= <comp>+
<comp>   ::= <block>
           | (repeat num <comp>+)
<block>  ::= <row>+
<row>    ::= <knot>+
<knot>   ::= RR | LL | RL | LR | SKIP
<num>    ::= n in Z+ (aka. any positive integer)
  
```

6 Semantics

See the next page for a table that summarizes the semantics of our program.

- What are the primitive kinds of values in your system? For example, a primitive might be a number, a string, a shape, a sound, and so on.

Syntax	Abstract Syntax	Type	Prec./Assoc.	Meaning
n	n of int	int	n/a	N is any positive integer. A primitive used in the repeat and setup operations (see below).
"A", "B", "C", "D", "_"	Knot = RR LL RL LR SKIP of char	char	n/a	Knots are primitives. We use these five chars to represent the four valid knots and the skip knot in the pattern.
"ABCD_"	Row of Knot list	Knot list	n/a	Rows are lists of knots. Since knots are basically chars, rows are sequences of chars, or strings.
"AAA BBB"	Block of Row list	Row list	n/a	Blocks are lists of rows. Blocks are a type of Comp, short for component.
(repeat n X X ... X)	RepeatOp of int * Comp list	int \rightarrow Comp list \rightarrow Comp list	1/left	The repeat operation will process the specified Comp n times. RepeatOp itself is also a type of Comp.
(name "Name")	NameOp of string	string \rightarrow string	1/left	The name operation gives the current pattern the name specified by the specified string.
(strings n "color" "color" ... "color")	SetupOp of int * string list	int \rightarrow string list \rightarrow string list	1/left	The setup operation allows the user to specify the number and colors of strings to be used in the pattern. Colors are inputted as strings.

Table 1: Semantics of our language.

Primitives could be a string representing the name of the pattern, the colors for each of the strings, and the four types of valid knots (or five if you count the SKIP option). The numbers used to specify how many strings there are or how many times to repeat something are also primitives.

- ii. What are the “actions” or compositional elements of your language? In other words, how are values used? For example, your system might have “commands” like “move pen” or compositional elements like “sequence of notes.”

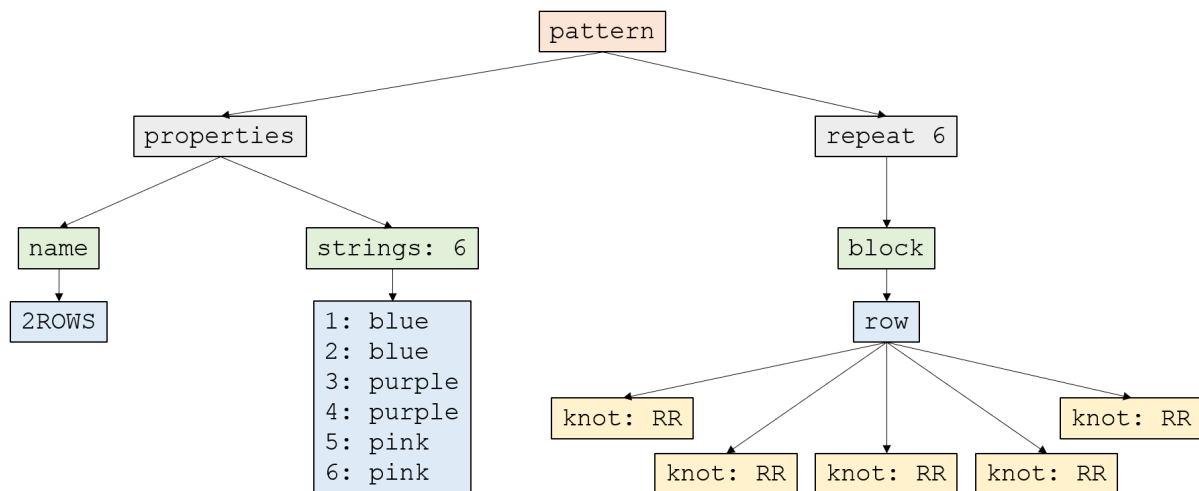
The main compositional elements are the rows of knots and blocks of rows. Some actions might be: “(name XXX)” to name the pattern, “(strings num color_1 color_2 ... color_n)” to specify the number and colors of strings, and “(repeat num XXX)” to repeat rows/blocks of the pattern. None of these operations cannot be combined with each other (but you can repeat a repeat operation), so precedence is not an issue.

- iii. How is your program represented? If it helps you to think about this using ML algebraic data types, please use them. Otherwise, rough sketches such as class hierarchy drawings are fine.

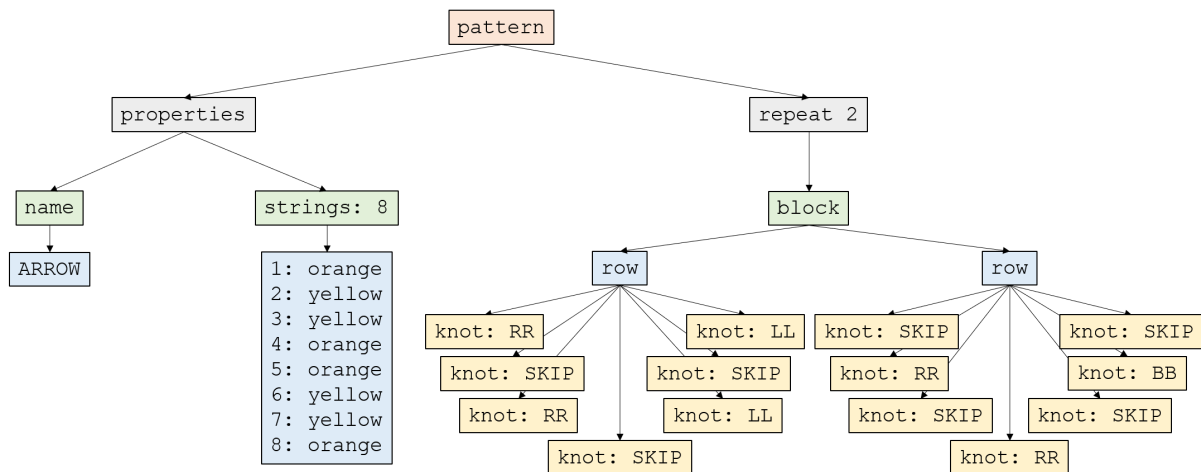
The program could be represented with a hierarchy similar to what’s depicted in the syntax diagram above. The name of the pattern and number/colors of the strings could be saved as variables. A row could be represented as a list of “knots” (after declaring this type) and a block could be represented as a list of “rows” (after declaring this a type too). The pattern would then be made up of blocks and repeat operations, which could be represented as a tuple of (1) the number of times to repeat and (2) the thing to repeat (ex. (2, row: AAAA)).

- iv. How do elements “fit together” to represent programs as abstract syntax? For the three example programs you gave earlier, provide sample abstract syntax trees.

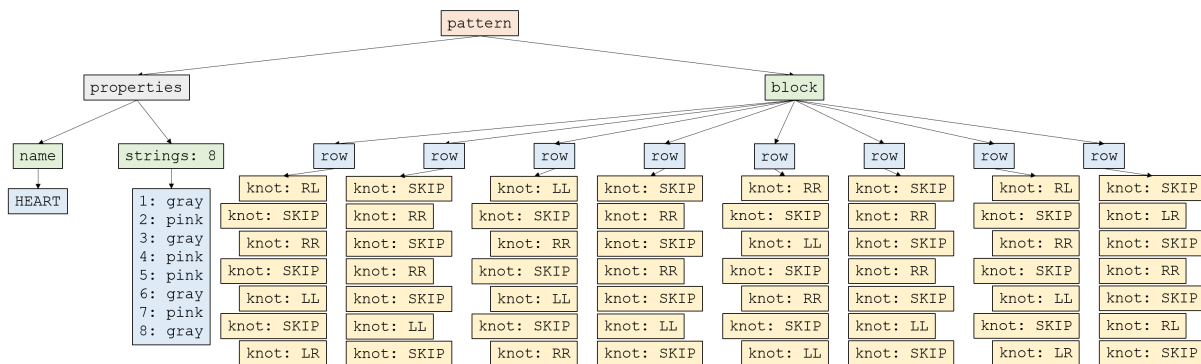
AST for Example 1:



AST for Example 2:



AST for Example 3:



Note: All knots are at same level, not at different levels, but to save space, they are stacked in this diagram.

v. How is your program evaluated? In particular,

- Do programs in your language read any input?

Not directly (at least with this current plan), but technically the name of the pattern, number of strings, and their colors are the input. These could be read in as parameters via the command line or specified by the user explicitly in their code later on. Maybe we could accept the code as a file input so the user doesn't have to type everything on the command line.

- What is the effect (output) of evaluating a program?

The output will be the pattern created by the user! Ideally, it will be displayed graphically on the screen after their program is complete. We currently plan to generate a svg from the user's program which they can open as a html file.

- Evaluation is usually conceived of as a depth-first, post-order traversal of an AST. Describe how such a traversal yields the effect you just described and provide illustrations for clarity. Demonstrate evaluation for at least one of your example programs.

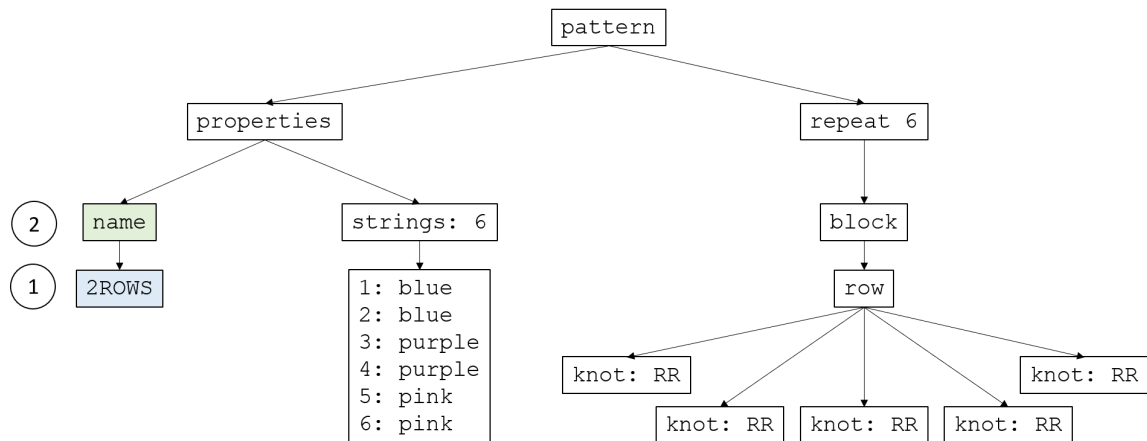
A depth-first, post-order traversal of the AST would yield the output described above because this would essentially first evaluate the name and number/colors of strings in the properties subtree, which is needed to evaluate the rest of the pattern. Then, we would evaluate the knots in each row, then the rows in each

block, then the blocks themselves, at which point the pattern can be produced since the blocks subtree is now evaluated in full too.

Here is how evaluation would work for example program 1:

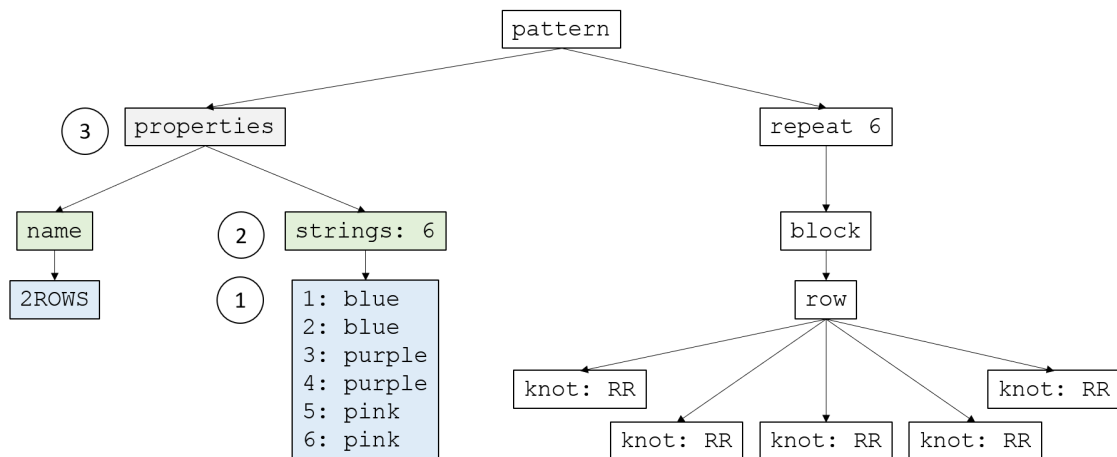
Step one:

1. (name 2ROWS)

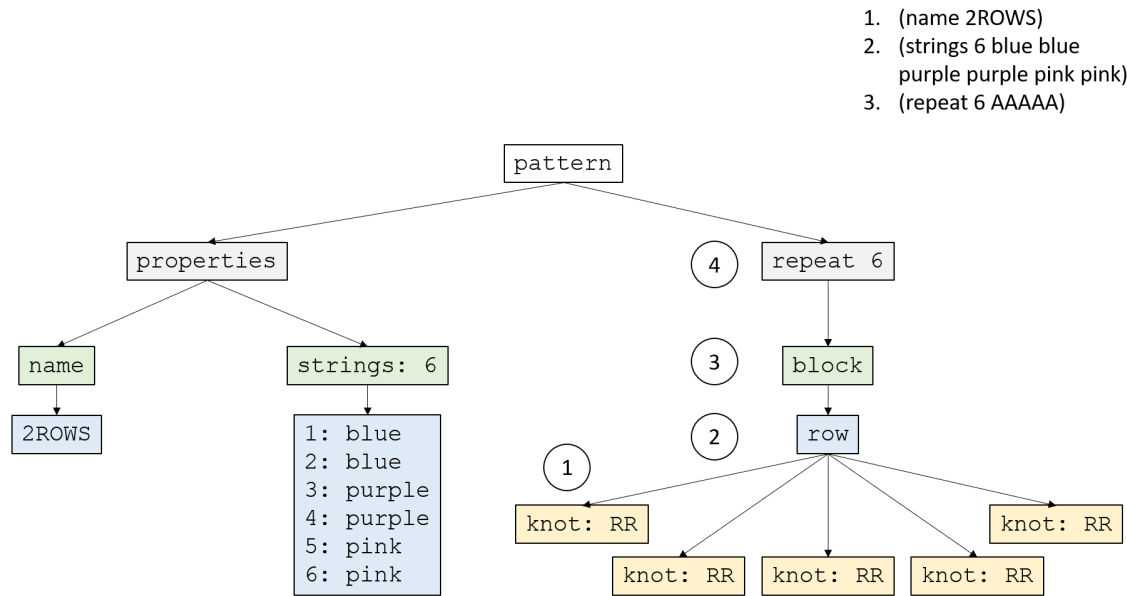


Step two:

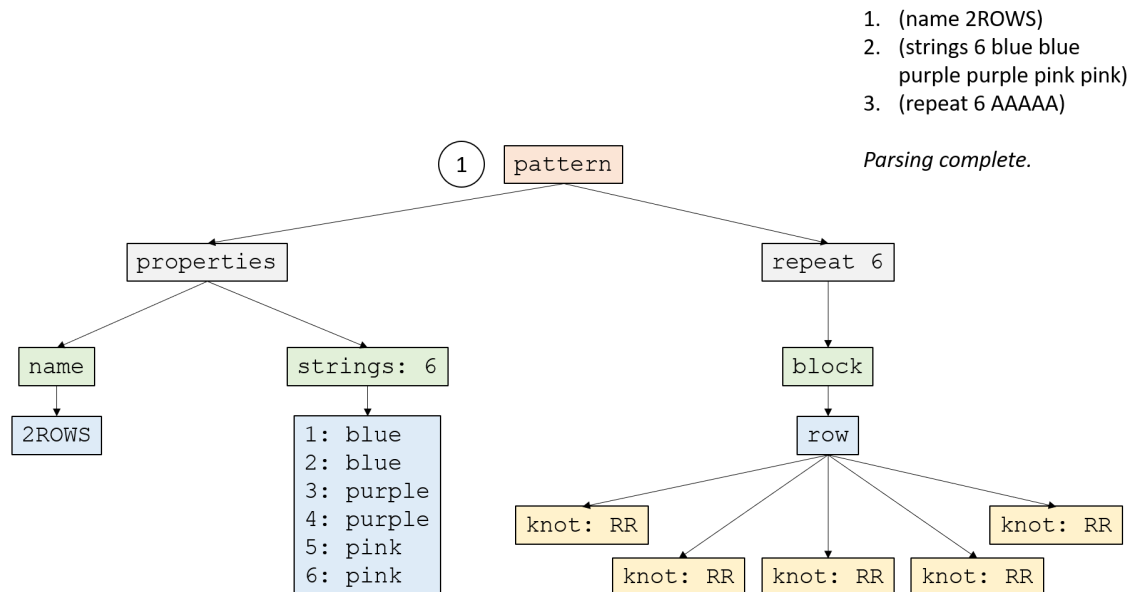
1. (name 2ROWS)
2. (strings 6 blue blue purple purple pink pink)



Step three:



Step four:



7 Remaining Work

We still need to implement the name/strings operations, which will probably require adding to the parser/interpreter code. Also we might try to allow new line characters too and file input so the user can just parse a file instead of typing everything out on the command line. As for interpreting the programs, we will continue working on converting the output to svg files and enhancing the resulting graphics.