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 so that anyone, regardless of coding experience, can design with it.

2 Design Principles

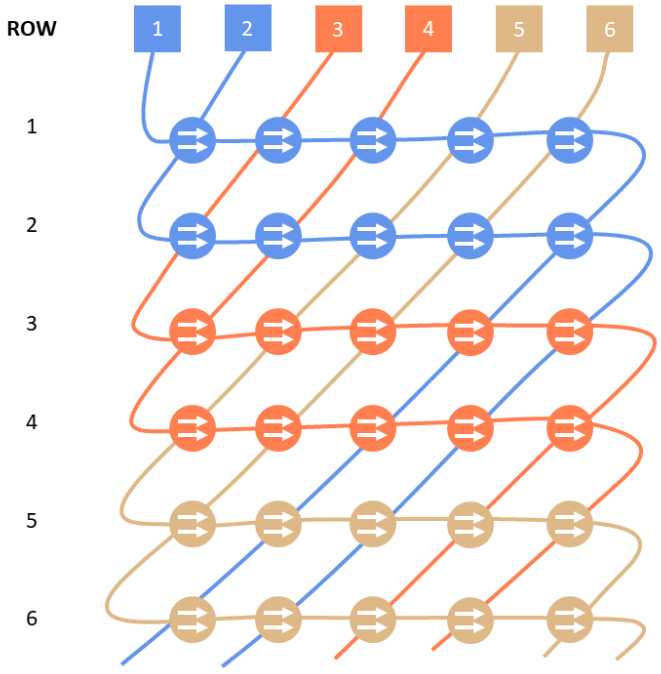
This language would definitely be guided by simplicity and clarity to keep the non-technical user in mind. We would also want the language to translate naturally from the physical art of bracelet making (ex. for a double right knot, maybe the user would type DR or something). The language would be flexible 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

To run these example programs, type `dotnet run ../../examples/example-1.fbp` on the command line from inside the lang directory. Replace "1" with "2" or "3" to try the other examples. The code and output is shown below. An SVG file matching the name of the pattern is created in the output directory in the main directory, which can be opened in a browser to produce an image.

Example 1: Two row pattern



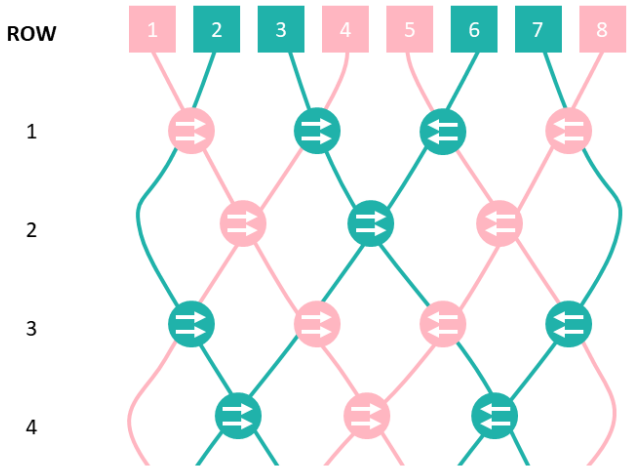

```
(name 2ROWS)

(strings 6 CornflowerBlue
CornflowerBlue Coral Coral
BurlyWood BurlyWood)

(repeat 6 AAAAA)
```

Evaluator output:

```
Pattern Name: 2ROWS

Row
1   CornflowerBlue >> CornflowerBlue >> CornflowerBlue >> CornflowerBlue >>
    CornflowerBlue >>
2   CornflowerBlue >> CornflowerBlue >> CornflowerBlue >> CornflowerBlue >>
    CornflowerBlue >>
3   Coral >> Coral >> Coral >> Coral >> Coral >>
4   Coral >> Coral >> Coral >> Coral >> Coral >>
5   BurlyWood >> BurlyWood >> BurlyWood >> BurlyWood >> BurlyWood >>
6   BurlyWood >> BurlyWood >> BurlyWood >> BurlyWood >> BurlyWood >>
```

SVG file output:

Example 2: Arrow pattern

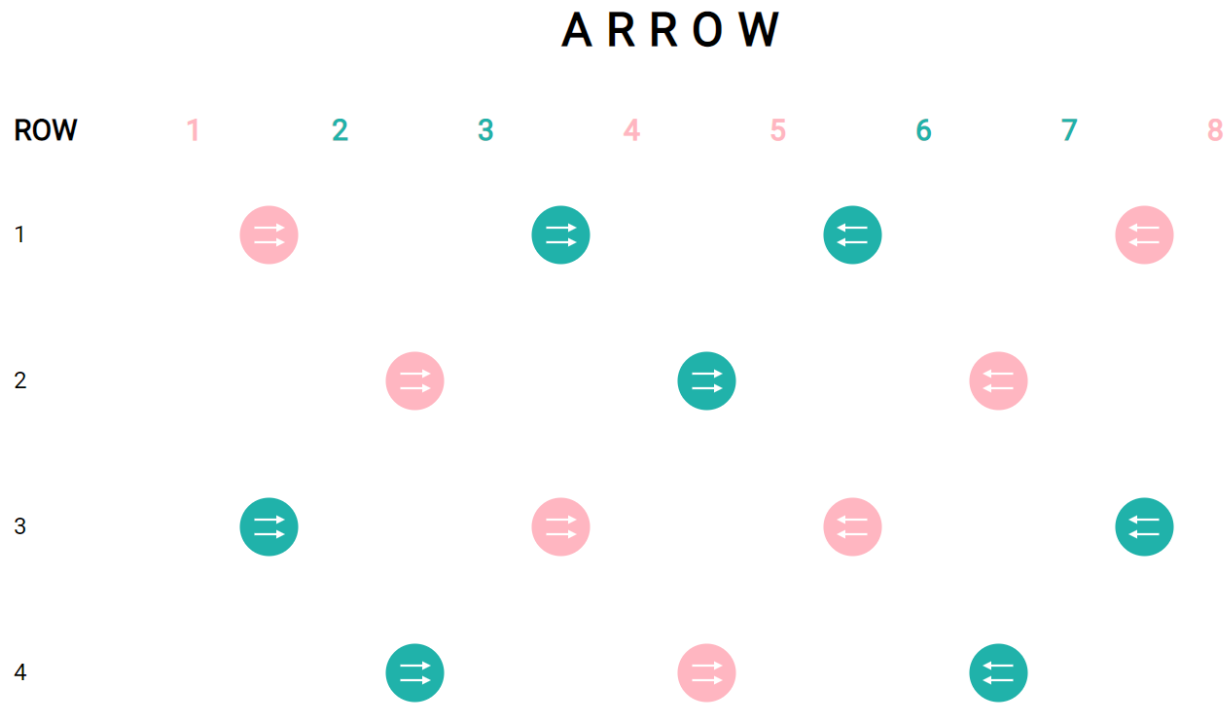



```
(name ARROW)

(strings 8 LightPink
LightSeaGreen LightSeaGreen
LightPink LightPink
LightSeaGreen LightSeaGreen
LightPink)

(repeat 2 A_A_B_B _A_A_B_)
```

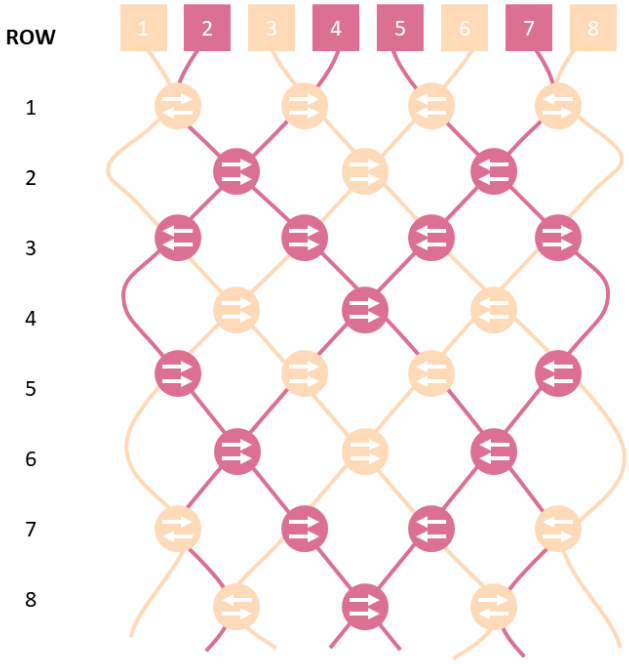
Evaluator output:

```
Pattern Name: ARROW

Row
1   LightPink >>  _ LightSeaGreen >>  _ LightSeaGreen <<  _ LightPink <<
2    _ LightPink >>  _ LightSeaGreen >>  _ LightPink <<  _
3   LightSeaGreen >>  _ LightPink >>  _ LightPink <<  _ LightSeaGreen <<
4    _ LightSeaGreen >>  _ LightPink >>  _ LightSeaGreen <<  _
```

SVG file output:



Example 3: Heart pattern

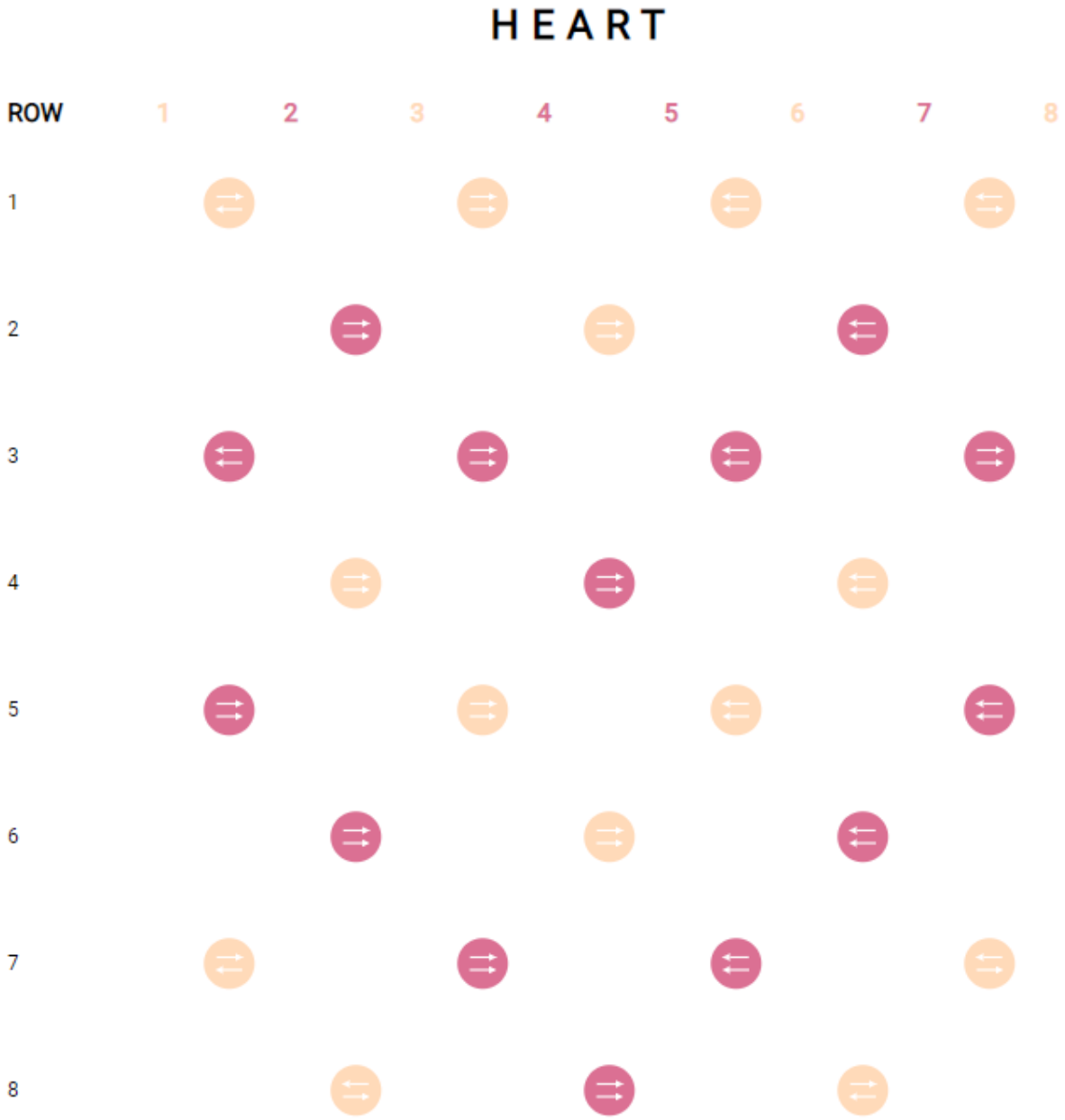



```
(name HEART)

(strings 8 PeachPuff
PaleVioletRed PeachPuff
PaleVioletRed PaleVioletRed
PeachPuff PaleVioletRed
PeachPuff)

C_A_B_D
_A_A_B_
B_A_B_A
_A_A_B_
A_A_B_B
_A_A_B_
C_A_B_D
_D_A_C_
```

Evaluator output:

```
Pattern Name: HEART

Row
1   PeachPuff >  _ PeachPuff >>  _ PeachPuff <<  _ PeachPuff <
2    _ PaleVioletRed >>  _ PeachPuff >>  _ PaleVioletRed <<  _
3   PaleVioletRed <<  _ PaleVioletRed >>  _ PaleVioletRed <<  _ PaleVioletRed >>
4    _ PeachPuff >>  _ PaleVioletRed >>  _ PeachPuff <<  _
5   PaleVioletRed >>  _ PeachPuff >>  _ PeachPuff <<  _ PaleVioletRed <<
6    _ PaleVioletRed >>  _ PeachPuff >>  _ PaleVioletRed <<  _
7   PeachPuff >  _ PaleVioletRed >>  _ PaleVioletRed <<  _ PeachPuff <
8    _ PeachPuff <  _ PaleVioletRed >>  _ PeachPuff >  _
```

SVG file output:



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.

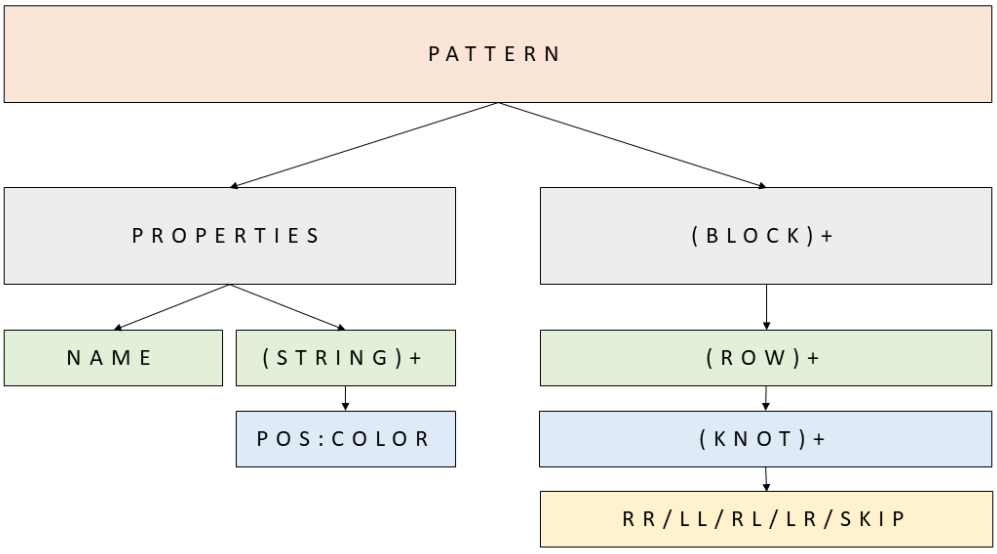
Each bracelet also has a specified number of strings, each of which can be any color. The strings have an original order to start the pattern, but usually will move around as rows are constructed (based on which knots are used and in what order). See example programs above for a basic visualization of 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").
- In short, a pattern must contain a **name**, **strings** (with assigned colors), and at least 1 **row** of knots.

A diagram summarizing the syntax is given below.



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

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

6 Semantics

i. 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.

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

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 for more details).
”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 C C ... C)	RepeatOp of int * Comp list	int * 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	1/left	The name operation assigns the current friendship bracelet pattern the name specified by the given string.
(strings n ”color” ”color” ... ”color”)	SetupOp of int * string list	int * 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.

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 can be combined with each other (but you can repeat a repeat operation), so precedence is not an issue.

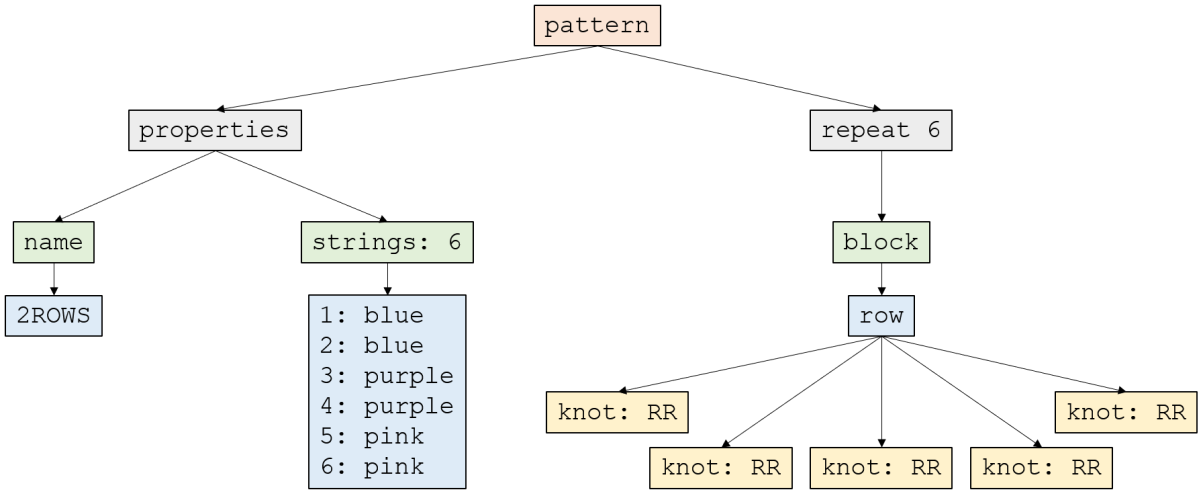
iii. How is your program represented? If it helps you to think about this using ML algebraic data types, please use them. Otherwise, rough sketches such as class hierarchy drawings are fine.

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)).

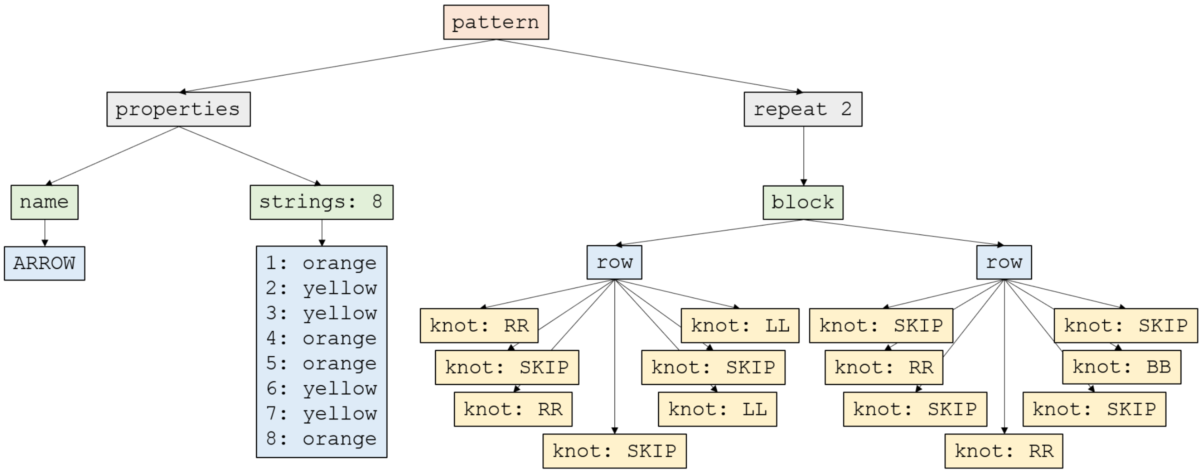
iv. How do elements "fit together" to represent programs as abstract syntax? For the three example programs you gave earlier, provide sample abstract syntax trees.

See below for sample ASTs. Note the examples from earlier have different colored strings, but their patterns and overall ASTs are otherwise identical to these.

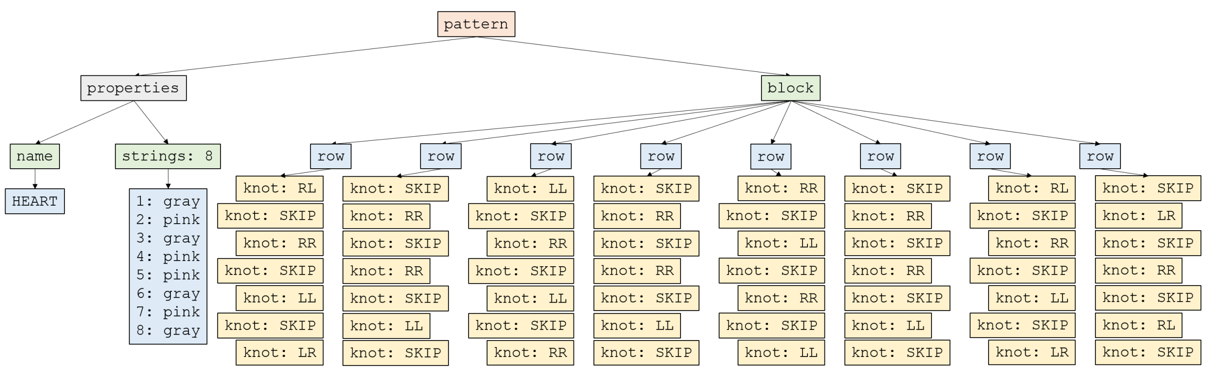
AST for Example 1:



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?

  The programs themselves can be inputted as files so the user doesn't have to type everything on the command line. Otherwise, there isn't any input read in by our programs.

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

  The output will be the pattern created by the user. Currently, the language outputs an SVG file generated by the user's program, which they can then open as an html file.

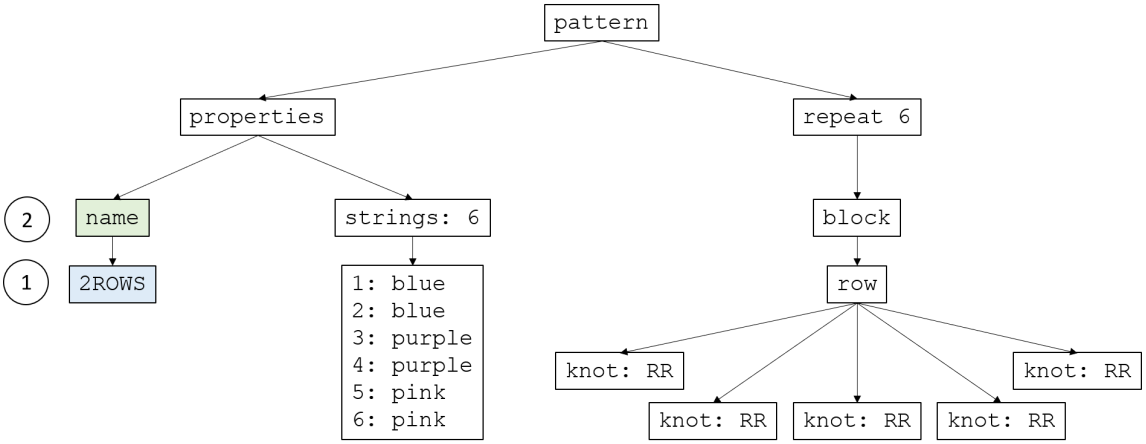
- Evaluation is usually conceived of as a depth-first, post-order traversal of an AST. Describe how such a traversal yields the effect you just described and provide illustrations for clarity. Demonstrate evaluation for at least one of your example programs.

  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 (again, the colors are different here, but nothing else should be impacted):

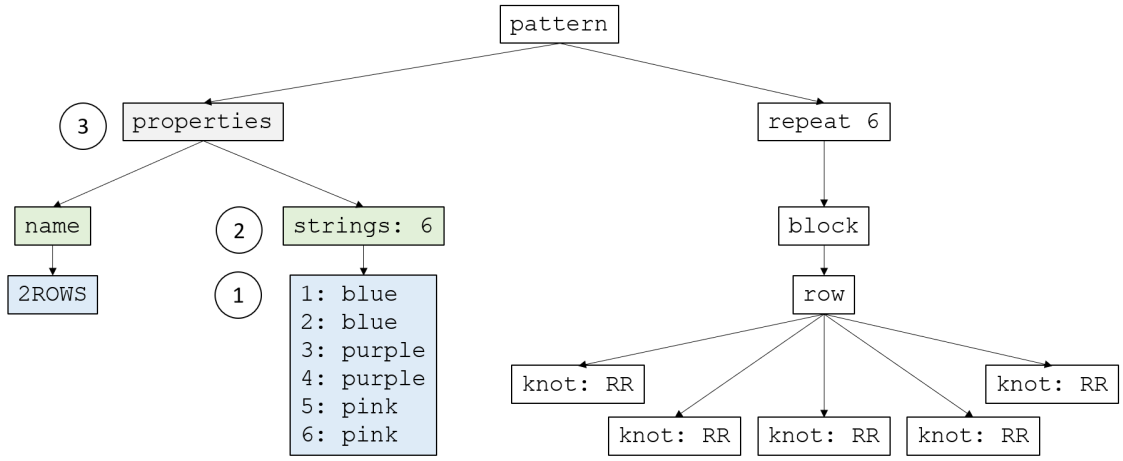
Step one:

1. (name 2ROWS)

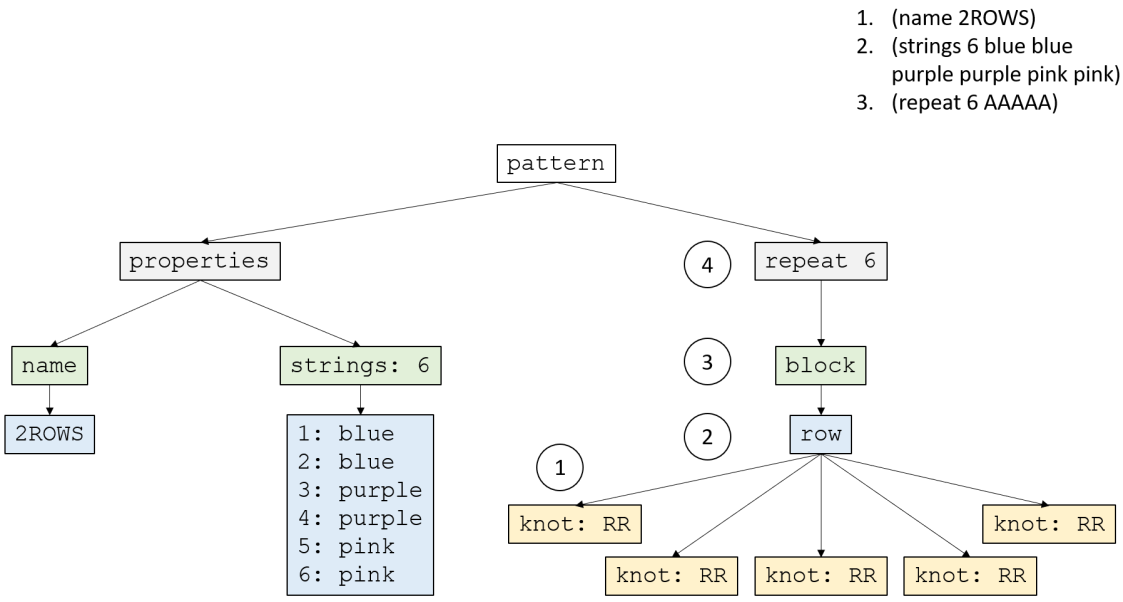


Step two:

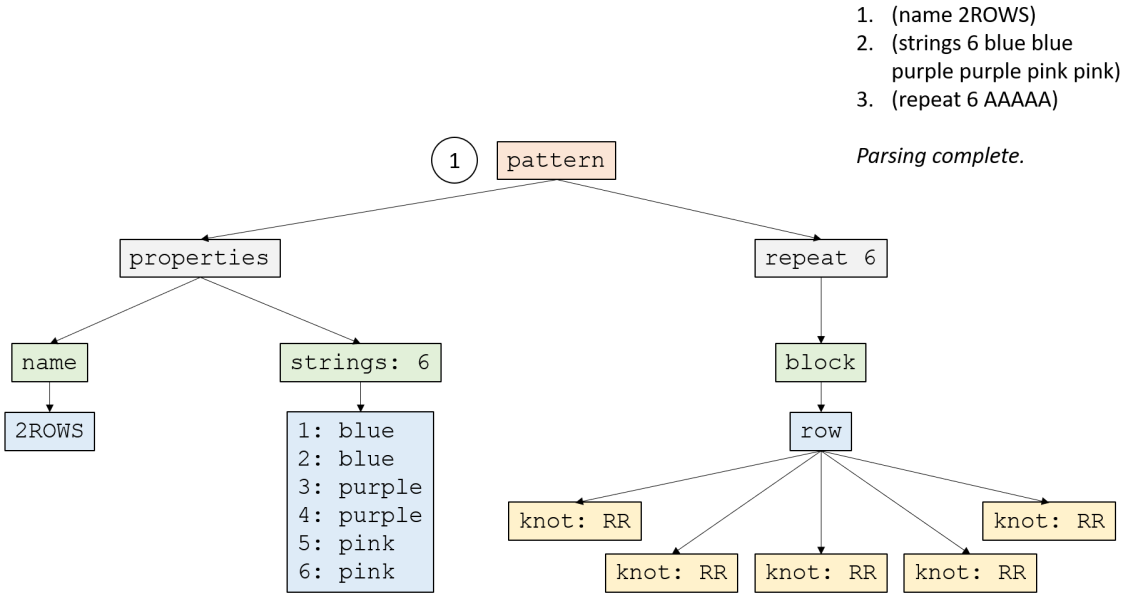
1. (name 2ROWS)
2. (strings 6 blue blue purple purple pink pink)



Step three:



Step four:



7 Remaining Work

We have finished implementing all the necessary operations and data types for our language, and our output is converted to SVG files. For a challenge, we are considering adding more features to safeguard the user against unsafe operations; for example, we can make sure that the user puts the correct number of knots in each row, which should be the number of strings minus one, by producing the correct error messages. We might also consider enhancing the graphics to include the strings along with the knots.