KnitPick#++ Specification

Locke Meyer, Felix King

0.1 Introduction

KnitPick#++ turns the tedious process of creating a knitting pattern, a set of instructions that describe the construction process of a specific garment, into a precise, painless, and intuitive programming experience. Ordinarily, creating a knitting pattern requires the graphic design savvy to organise the document cleanly and clearly; with KnitPick#++, this are no longer required. Furthermore, the various styles of knitting pattern that exist can be confusing to a knitter without a wide range of prior knitting experience; KnitPick#++ establishes a powerful and expressive standard for knitting patterns that can be easily learned and read by anyone.

Many knitting patterns rely not only on repetition of single stitches, but also entire rows of stitches and of sets of rows. Also, since a secondary goal of knitting patterns is for a given pattern to produce the exact same garment every time, the output is deterministic. For these reasons, the problem of generating knitting patterns is particularly suited to a programming language-based solution, since programming languages offer easy repetition of pieces of information as well as precision.

0.2 Design Principles

One of our technical goals for KnitPick#++ is for it to be relatively lightweight in terms of (e.g.) the number of basic stitches that are defined by the language. However, we also want to design KnitPick#++ such that users can define new stitch types to fit their needs – for instance, by defining a less common stitch that serves a more niche purpose in the knitting process, like the moss stitch. Another one of our goals for KnitPick#++ is to have it generate documents that look nice and are easy to comprehend. An aesthetic goal is for KnitPick#++ code to be concise and readable; for this reason, many keywords in our syntax are abbreviations.

0.3 Examples

Each of the following examples can be executed from within the examples folder by typing the following into the terminal: ”dotnet run –project ../../code/kpplibrary example-1.kpp example-1”. Here, ”example-1.kpp” is the name of the file containing KnitPick#++ code and ”example-1” is the name of the pdf file that will be generated.
KnitPick#++ will automatically create a pdf of your knitting pattern.
A.

```
TODO
```

B.

```
TODO
```

C.

```
TODO
```

0.4 Language Concepts

The user will be thinking in terms of **paragraphs** consisting of text instructions. Each paragraph of text will correspond to a part of the garment – for instance, the ribbing of a scarf or the cuff of a sweater. Furthermore, each paragraph of text is made up of **instructions**, and these instructions are themselves made up of **sequences** including **stitches**, numbers, or other more specialized directives. For example, one paragraph of a sock knitting pattern might look like this.

```
Cuff
Step 1. cast on 64.
Step 2.
    (a) knit 2, purl 2 to end of row.
Repeat for a total of 16 times.
```

The primitives of this language are integers, floats, and strings. These primitives will all combine in various ways to create the paragraphs of text instructions in the final pattern. Integers are used to represent the number of repetitions of a stitch and the number of times to repeat an instruction, among other things. Strings represent the names of stitches and any custom instruction the user wants to include. Floats appear in the header of a document and provide precise information about the ”gauge” of the pattern and the size of the yarn to be used.

0.5 Formal Syntax

The start expression is < program >.

```
<program>     ::= (<assignment>)*<document>
<assignment>  ::= "let "<variable>" ="<pad><string><pad>
<variable>    ::= (<letter>)+
<document>    ::= <pad><header><pad>(<pad><paragraph><pad>)+
<header>      ::= "header "<pad><string><pad><needle><pad><gauge><pad><yarn>
<needle>      ::= "needle "<pad>"sp"<pad><string><pad><integer>
                |   "needle "<pad>"dp"<pad><string><pad><integer>
                |   "needle "<pad>"ci"<pad><string><pad><integer>
<gauge>       ::= "gauge "<oParen><float><pad><comma><pad><float><cParen>
<yarn>        ::= "yarn "<pad><string><pad><integer>
<paragraph>   ::= "pg "<string>(<pad><instruction><pad>)+
<instruction> ::= <string>
                |   <row>
<row>         ::= (<stitchSeq><pad>)+
<stitchSeq>   ::= <stitch><integer>
                |   <plus><stitchSeq>
                |   <oParen><stitchSeq><stitchSeq><cParen>
<stitch>      ::= 'k'
                |   'p'
                |   <variable>
<pad>         ::= (<ws>)*
<string>      ::= <quote>(<character>)+<quote>
<integer>     ::= (<d>)+
<float>       ::= (<d>)+<dot>(<d>)+
<d>           ::= '0' | '1' | ... | '8' | '9'
<period>      ::= '.'
<plus>        ::= '+'
<oParen>      ::= '('
<cParen>      ::= ')'
<comma>       ::= ','
<ws>          ::= ' ' | '\n' | '\t' | "\rn"
<quote>       ::= '"'
<letter>      ::= 'A' | ... | 'Z' | ... | 'a' | ... | 'z' |
<character>   ::= ' ' | '!' | ... | 'a' | ... | 'z' | ... | '}' | '~'
```

0.6 Semantics

Syntax	Abstract Syntax	Type Syntax	Prec./assoc.	Meaning
v	`Var of String`	`string`	n/a	*v* represents a variable, which can store a stitch. Every use of the variable gets replaced with the stitch it represents.
”let ”*var*” = ”*st*	`Assignment of Var * String`	`Assignment`	n/a	An Assignment associates a var with a string for later use.
n	`Int of int`	`int`	n/a	*n* is a primitive. We represent integers using the 32-bit F# integer data type (int32).
”*str*”	`String of string`	`string`	n/a	*str* is a primitive. We represent strings using the built-in F# string type.
’k’ or ’p’	`Stitch of String * Int`	`string * int`	n/a	’k’ or ’p’ represents either a knit or a purl stitch. We construct a Stitch object by keeping a (small) dictionary of stitches and their abbreviations.
sn	`StitchRep of Stitch * Int`	`(string * int) * int`	n/a	*sn* is a combining form that communicates a particular kind of stitch (knit or purl) and the number of times it should be performed.
+*S*	`StitchSeqToEnd of StitchSeq`	`StitchSeq`	n/a	+*S* is a kind of StitchSeq that indicates that the StitchSeq *S* should be repeated until the end of the row.
(*SS*)	`TwoStitchSeq of StitchSeq * StitchSeq`	`StitchSeq * StitchSeq`	n/a	(*SS*) is a combining form that allows StitchSeqs to be chained for more complexity.
R	`Row of StitchSeq list`	`StitchSeq list`	n/a	A row combines multiple stitch sequences that, together, make up one row of knitting.
I	`Instruction`	`Instruction`	n/a	*I* is an instruction, either a row of knitting or other direction (for example, ”cast on.”)

Syntax	Abstract Syntax	Type Syntax	Prec./assoc.	Meaning
”pg ”*name inst*	`Paragraph of String * Instruction list`	`string * Instruction list`	n/a	The string ”pg ” followed by a string and a list of instructions defines a paragraph. The string is the name of the paragraph and the list of instructions describes how that part of the garment should be knit.
”needle ” *ty sys size*	`Needle of String * String * Int`	`string * string * int`	n/a	The string ”needle” followed by two strings and an integer represents the needles needed for the knitting pattern. *ty* represents the type of needle, *sys* represents the measurement system (either US or mm), and *size* is the size in that system.
”gauge ” (*stitches, rows*)	`Gauge of Float * Float`	`float * float`	n/a	The string ”gauge ” followed by a tuple of floats in parentheses communicates the ratios of stitches to horizontal inches and rows to vertical inches that the pattern designer used.
”yarn ” *”type” size*	`Yarn of String * int`	`string * int`	n/a	”yarn ” followed by a string and and integer represents the type / brand and the size of the yarn used in the pattern
”header ” *name needle gauge yarn*	`Header of String * Needle * Gauge * Yarn`	`Header`	n/a	The header data type is required in each pattern and contains the information about the needles, gauge, and yarn.
header pgs	`Document of Header * Paragraph list`	`Document`	n/a	A Document consists of a Header followed by a list of paragraphs.
assigns doc	`Program of Assignment list * Document`	`Assignment`	n/a	A program wraps a set of assignments and the Document datatype in that order.

0.7 Remaining Work

In our previous draft, we had mentioned wanting to implement a Taper type – this actually turned out to be redundant due to the Repeat type. We also mentioned that we wanted to implement variables. And we did! HOWEVER, due to the design of our whole abstract syntax tree, we realized that we wouldn't be able to implement a variable that could represent any arbitrary data type. This was a valuable learning experience! We did end up implementing variables anyway; it's just that they can only represent stitches.

In the future, it would be nice to add a way for the user to create a colored grid pattern as a visual aid to the knitting instructions. This was something we had originally thought about doing. However, we realized early on in the process of designing KnitPick#++ that we just wouldn't end up having time to implement it, but it could be cool to come to and work on at some point.