The KnitML User's Guide

Jonathan Whitall

The KnitML User's Guide

Jonathan Whitall

Table of Contents

List of Tables

Part I. Getting Started

Table of Contents

Chapter 1. Introduction

Overview

The KnitML project's goal is to define and promote the adoption of a standard content model for knitting patterns. By developing a community-supported specification and providing basic rendering and transformation tools, the project aims to make KnitML easy to use and valuable to the knitter.

Knitting patterns are algorithms as much as they are documents. Consequently, it is important that a computer to be able to understand the algorithm so that it can make decisions. At the same time, a pattern must be able to be rendered into a published pattern in human-readable form. A KnitML pattern, therefore, is a formalization of the algorithm required to produce a knitted object, composed in a way that can still be rendered as a comprehensible published pattern.

The tools provided by the KnitML Project are meant to show the value of KnitML to the knitting community by demonstrating the advantages to standardizing on a content model. They are by no means exhaustive. Independent efforts to write software which understands KnitML patterns (such as the Knitter project) are already underway.

Pattern Formats

KnitML patterns are expressed primarily using a technology called XML (Extensible Markup Language). XML is an international standard for expressing custom document markup. KnitML provides validation rules to an XML parser (in the form of a schema) which ensures that the knitting pattern follows the expected syntax. If you use an XML authoring tool to develop KnitML patterns, the software's suggestions and completion tools will help you more quickly gain an understanding of the XML syntax.

XML is well suited to serve as an interchange format for software, though it is not without its disadvantages. For one, it's not a particularly compact syntax, nor does it flow, read, or even resemble a traditional knitting pattern. While XML is a language that is human readable, it is not intended to be directly edited by humans. Ideally, a KnitML authoring tool would do this and make pattern designers' lives easier.

The Knitting Expression Language, or KnittingEL, attempts to provide a far easier way to express a KnitML pattern. A KnittingEL pattern uses familiar knitting pattern language constructs and is more free form in nature. As a result, it looks more like a published knitting pattern than its XML counterpart. KnittingEL is not intended to be a software interchange format; it is a pattern authoring tool only. Patterns written in KnittingEL are converted to XML before passed to supporting software (such as the validator and the pattern renderer).

While we encourage all patterns to be developed in KnittingEL, tooling support is limited to basic text editing. Because of this, it's pretty easy to create KEL files that will produce invalid XML files. XML tooling is far more extensive and will better be able to guide the pattern designer to create a valid pattern.

Known Limitations

- You cannot currently express seaming knitted edges together
- Provisional cast ons are not yet supported
- Support for picking up stitches is somewhat limited (because edges cannot be expressed)
- n-sized decreases are not yet supported (where n is more than a double decrease)

- repeats are completely literal (there is no way to express a soft repeat)

Chapter 2. Installing and Running KnitML

Installation

KnitML requires Java 1.5 (or higher) to run. If you do not have Java 1.5 installed, please visit the Sun website to download it (http://java.sun.com).

To set up your machine to run the `knitml` executable, you must:

1. Set the `JAVA_HOME` environment variable to the root directory of your Java installation (for example `C:\Program Files\Java\jre1.5.0_11`)

2. Set the `KNITML_HOME` environment variable to the root directory of where you unzipped the KnitML distribution (for example C:\Program Files\knitml-0.4).

3. For Windows, set the `KNITML_HOME/bin` directory in your PATH environment variable so that the `knitml` command can be run from any directory. (The actual value will be `%KNITML_HOME %/bin` appended to the end of the PATH variable.)

For Windows XP, you can set environment variables from the Start menu. Select Settings, then Control Panel, then System. Select the Advanced tab, then press the Environment Variables button, highlight the System Variable (or User Variable) with the name you want, click Edit, add or append the desired value, then click OK twice. (Whew!)

Project Layout

The `samples` directory contains several different patterns to showcase the both XML and KnittingEL patterns:

- A simple square (for use with the tutorial in this document)
- A sampler swatch (includes color work and cables)
- A basic sock
- A sock based from the Nutkin pattern (called Nutkin2)

The `conf` directory contains files which configure the way KnitML runs. It includes logging configuration as well as several files which configure the validator and pattern renderer. The conf directory is added to the classpath when a `knitml` command is executed.

The pattern renderer uses a set of property files (called a resource bundle) built into the module that allow rendering in different languages. Currently, KnitML provides support for both English and Spanish rendering. This will expand as people contribute more languages. If you would like to add support for a new language, see the section called “Adding Support for a New Language”.

You can use your own property file to further customize how a pattern will look when it is rendered. See the section called “Personalizing Patterns” for guidance.

Processing XML files

To process a KnitML file in XML format, first change to the directory where your XML file is located, then run one of the following commands:

- `knitml validate <xml-filename>`

```
                [-output <validated-filename>]
                [-checksyntax]
```

Creates a validated XML file from an unvalidated XML file

- `knitml render <xml-filename>
              [-output <pattern-filename>]
              [-checksyntax]`

  Renders a pattern from a validated XML file

- `knitml validateAndRender <xml-filename>
                          [-output <pattern-filename>]
                          [-checksyntax]`

  Renders a pattern from an unvalidated XML file

If the `-output` option is not specified, the result will be sent to the screen (i.e. "standard out"). The `-checksyntax` option performs XML schema validation against the provided file. You do not need to be connected to the Internet to check the syntax, as the software is configured to resolve the XML schemas provided with the KnitML core JAR file.

You can also use the following shortcuts: `r` for `render`, `v` for `validate`, `vr` for `validateAndRender`.

Processing KEL files

To process a KnitML file in KEL format, first change to the directory where the KEL file is located, then run the appropriate KnitML command:

- `knitml convert <knittingEL-filename> [-output <knitml-filename>] [-checksyntax]`

  Converts a KEL file to an unvalidated XML file

- `knitml convertAndValidate <knittingEL-filename> [-output <validated-knitml-filename>] [-checksyntax]`

  Converts a KEL file to a validated XML file

- `knitml convertValidateAndRender <knittingEL-filename> [-output <pattern-filename>] [-checksyntax]`

  Converts a KEL file directly to a rendered pattern

`-output` and `-checksyntax` behave as described in the previous section.

You can also use the following shortcuts: `c` for `convert`, `cv` for `convertAndValidate`, `cvr` for `convertValidateAndRender`.

Part II. Writing KnitML Patterns

Table of Contents

Chapter 3. The Knitting Expression Language

Overview

The Knitting Expression Language (formerly known as GroovyKnit) is based on the Groovy programming language and GroovyMarkup architecture (`http://groovy.codehaus.org/GroovyMarkup`). Elements are specified by name, and sub-elements go between curly braces following the name. So this in KnittingEL:

```
pattern {
  generalInformation
}
```

looks like this in XML:

```
<pattern>
  <general-information/>
</pattern>
```

`generalInformation` in KnittingEL became `general-information` in XML because `generalInformation` is a defined KnittingEL function that acts as an intermediary to the XML format to increase readability. In this case, it simply transforms its name to `general-information`, as you cannot use dashed names in KnittingEL without quoting them. So while you could also do the following to achieve the same result, it's not quite as readable.

```
pattern {
  'general-information'
}
```

See Table B.1, “Defined Expression Language Functions” for a complete list of KnittingEL functions.

You have the option to capitalize the first letter of any and all KnittingEL functions. So the following would produce the same XML:

```
Pattern {
  GeneralInformation
}
```

In many cases, this is more readable.

To specify an XML element's value as text, simply include it as a parameter. This:

```
row {
  knit (3)
}
```

Becomes:

```
<row>
  <knit>3</knit>
</row>
```

Note that any of the samples below will produce the same XML as above (this is not an exhaustive list of possibilities):

```
row {
```

```
  knit 3
}

Row {
  Knit 3
}

row { knit 3 }

row { k 3 }

row { k3 }

Row: k3
```

The first three examples show that line placement inside of curly braces isn't crucial, as long as the method name and its arguments (which occur before the curly braces) are on the same line[1]. The last three examples use k as a shorthand for knit. `k` is a defined KnittingEL function which expands to the `knit` element. The KnittingEL parser is smart enough to recognize that, since `k3` ends with a number, the number belongs as an argument to the `k` method rather than part of the method name.

To express several knitting operations in a row, you can write them in any of the following ways:

```
row {
  knit 3
  purl 3
}

row { knit 3, purl 3 }

row ( k3, p3 }

row: k3, p3

Row: k3, p3
```

And you'll get this:

```
<row>
  <knit>3</knit>
  <purl>3</purl>
</row>
```

Note that you can use the `:` syntax to define a row instead of the brace notation, in which case the entire row must be defined on the same line. If you have a more complex row to define that needs to be split into multiple lines, use the `{}`s instead.

You can use the : syntax for any of the following functions: `row`, `round`, `nextRow`, `nextRound`, `shortRow`. The `:` syntax used with other functions exists for language purposes only and is simply ignored (producing the same effect as if it were omitted).

KnittingEL also defines shorthands for common operations. Here are a few:

KnittingEL	**XML**
`row { k to end }`	`<row>` `  <repeat until="end">`

[1]Even parameters can be on a different line if parentheses are explicitly specified. Map arguments delineated by [] can also be split across lines. However, a row which uses the colon syntax must be completely defined on the same line.

KnittingEL	XML
	<knit/> </repeat> </row>
round { k5 }	<row type="round"> <knit>5</knit>
shortRow { k5, turn }	<row short="true"> <knit>5</knit> <turn/> </row>

An element's attributes appear before the curly braces in a series of name/value pairs in a bracket (called a Map):

```
yarnType [brand:'Cotton Classic', weight:'worsted'] { ... }
```

It's very common for an element to have an ID attribute, so there is often a function shorthand. For example, the following three lines are equivalent:

```
yarnType 'my-yarn' [brand:'Cotton Classic', weight:'worsted']
yarnType [id:'my-yarn', brand:'Cotton Classic', weight:'worsted']
<yarn-type id="my-yarn" brand="Cotton Classic" weight="worsted"/>
```

Writing a Simple Pattern

For this example, we will compose the KnitML pattern using the KnittingEL syntax. The pattern is for a simple 4x4 in square knitted in stockinette stitch.

Create a file name `simple-square.kel`. We will start by defining the `pattern` element, the root element for all KnitML patterns. Since we're not internationalizing this pattern, we'll specify that it was written in `'en'` (the ISO-639 Language Code for English).

```
Pattern 'en' {}
```

Next, we'll add some header information about the pattern to a `generalInformation` element within the `pattern` element.

```
Pattern 'en' {
  GeneralInformation {}
}
```

Let's give the pattern a name. Name, description, author, etc. go under `generalInformation`. Anything that should be interpreted as a literal value (for instance, a string of text) should be in single quotes.

```
Pattern 'en' {
  GeneralInformation {
    Name: 'Simple Square'
    Description: 'A simple square'
    Dimensions: '4 in by 4 in'
  }
}
```

Gauge also goes here. We'll indicate that the gauge swatch should be knit in stockinette at 6 stitches to the inch and 6 rows to the inch.

```
Pattern 'en' {
  GeneralInformation {
```

```
    Name: 'Simple Square'
    Description: 'A simple square'
    Dimensions: '4 in by 4 in'
    Gauge 'stockinette' {
      StitchGauge: 6 stitchesPerInch
      RowGauge: 6 rowsPerInch
    }
  }
}
```

The colon after the function name is optional. It increases readability in this case[2].

Let's move on to the next section. We also need some supplies to knit this pattern. Generally a knitter needs needles and yarn, and needs to know what type of each the pattern calls for. In this case, we're not particularly picky about the yarn (as long as it is plausible for obtaining gauge). The only thing we are going to specify is that the weight should be worsted. Likewise, we are going to specify using a size US 4 circular needle of any length.

```
Pattern 'en' {
  GeneralInformation { ... }
  Supplies {
    Yarns {
      YarnType 'the-yarn' [weight:'worsted']
    }
    Needles {
      needleType 'size-4' circular { size: 4 US }
      needle 'the-needle' [typeref:'size-4']
    }
  }
}
```

First we added a `yarns` sub-element to the supplies element. This is where the yarns that will be used in the pattern are defined. Both yarn types (e.g. brand, weight, etc.) and yarn colors of a specific type may appear. In this case, we are only using one type of yarn (with an identifier of 'worsted-weight-yarn' so that we can refer to it later in the pattern if need be). As we don't care about color, there are no `yarn` elements to define.

Similarly, we have a needles element where all of the needles that will be used in the pattern are defined. Both needle types (e.g. DPN, circular, and other attributes) and number of needles of a particular needle type must be defined. Here we define one needle type we're identifying as 'size-4'. It's a circular needle ('circular') and its size is US 4. We also define that we need one needle of type 'size-4' and we're going to call that needle 'the-needle' (since this project only uses one needle).

Now we will add directions.

```
Pattern 'en' {
  ...
  Directions {}
}
```

Since our gauge is 6 stitches to the inch and we want to knit a 4-inch square, the first thing to do is to cast on 24 stitches (6 times 4). You can specify a cast-on method if desired. In this case, we'll specify that the knitter should use the long-tail cast on.

```
Pattern 'en' {
  ...
  Directions {
    CastOn 24 'long-tail'
```

[2]Note that using a colon here has a slightly different implication than using it with the `row` family of functions, as was previously discussed.

```
  }
}
```

Stockinette stitch is a two-row repeat when knitting flat. The first row is all knit, and the second is all purled.

```
Pattern 'en' {
  ...
  Directions {
    CastOn 24 'long-tail'
    Row 1: k to end
    Row 2: p to end
  }
}
```

Since the project's dimensions should be 4 inches tall as well as wide, we don't want to have to write out 22 more rows of the same two rows. So we will declare rows 1 and 2 to be an instruction, give that instruction an identifier ('stockinette-st'), and then repeat it a certain number of times. In this case, we will repeat the two stockinette rows until the project measures 4 inches. KnitML also calculates row numbers for you if not provided, so we will remove them.

```
Pattern 'en' {
  ...
  Directions {
    CastOn 24 'long-tail'
    Instruction 'stockinette-st' {
      Row: k to end
      Row: p to end
    }
    Repeat 'stockinette-st' until measures 4 in
  }
}
```

`until measures` are keywords which inform the KnittingEL processor to construct the particular XML form when an instruction is repeated to a measurement. There are various other keywords in the KnittingEL. See the appendix for a list of them.

Now that we've reached 4 inches, we will finish the project by binding off all stitches.

```
Pattern 'en' {
  ...
  Directions {
    CastOn 24 'long-tail'
    Instruction 'stockinette-repeat' {
      Row: k to end
      Row: p to end
    }
    Repeat 'stockinette-st' until measures 4 in
    Row: BindOff all sts
  }
}
```

That's all there is to it! The complete pattern looks like this:

```
Pattern 'en' {
  GeneralInformation {
    Name: 'Simple Square'
    Description: 'A simple square'
    Dimensions: '4 in by 4 in'
    Gauge 'stockinette' {
```

```
      StitchGauge: 6 stitchesPerInch
      RowGauge: 6 rowsPerInch
    }
  }
  Supplies {
    Yarns {
      YarnType 'the-yarn' [weight:'worsted']
    }
    Needles {
      NeedleType 'size-4' circular { size: 4 US }
      Needle 'the-needle' [typeref:'size-4']
    }
  }
  Directions {
    CastOn 24 'long-tail'
    Instruction 'stockinette-st' {
      Row: k to end
      Row: p to end
    }
    Repeat 'stockinette-st' until measures 4 in
    Row: BindOff all sts
  }
}
```

If you run this through the basic text rendering program, you'll get this:

```
Simple Square
A simple square

Stitch Gauge: 6 st/in
Row Gauge: 6 row/in

Yarn:
    worsted weight

Needles:
    1 circular needle size 4 US (3.5 mm)

Directions
----------
Using the long-tail method, cast on 24 stitches.
Row 1: Knit
Row 2: Purl
Repeat rows 1-2 for 4 in.
Row 25: Bind off all stitches
```

This may not appear to be worth all of the trouble, but keep a few things in mind. First of all, the pattern has been "test knit." This means If there were too many or two few stitches for any given row, the validator would raise an error. This can be very helpful because it can catch certain kinds of errata immediately. The validator also fills in missing information as it's being knit. Notice that all of the rows have been assigned numbers.[3]

Second, note that our needle size was automatically converted to metric. There are various options on the pattern renderer that will allow you to use the system of units most familiar to you, regardless of what the original pattern was written in. (Note that the pattern renderer does not currently support these options, but it will in a future release.)

[3]Also note that the bind off row is labeled as Row 25. That's because the pattern's row gauge is set to 6 rows per inch, and the repeat was done for 4 inches. That would make the next row knit row 25.

Here is the gauge specification in metric:

```
Simple Square
A simple square

Stitch Gauge: 2.4 st/cm
Row Gauge: 2.4 row/cm
...
```

Also note that, even though we specified that the pattern is written in English, this pattern can (for the most part) be rendered in another language very easily. Simply setting the renderer to use Spanish (using pattern conventions from Spain) produces this result:

```
...
Instrucciones
-------------
Usando el montado doble, monta 24 puntos.
Hilera 1: Todo del derecho
Hilera 2: Todo del reves
Repite hileras 1-2 hasta que la pieza mida 4 in.
...
```

Note that the header information (i.e. the parts of the pattern written in English) would always be written in English unless you internationalized them. See Chapter 5, *Internationalization* for more information.

Improving the Pattern

Our simple square pattern renders fine. There are, however, a couple of quirks that we can eliminate. The first is that there are not line breaks where you would expect them to be in a traditional pattern. You can fix this by adding section elements to break the pattern up into logical paragraphs.

```
Pattern 'en' {
...
    Section {
      CastOn 24 'long-tail'
    }
    Section {
      Instruction 'stockinette-st' {
        Row: k to end
        Row: p to end
      }
      Repeat 'stockinette-st' until measures 4 in
    }
    Section {
      Row: BindOff all sts
    }
  }
}
```

The renderer will produce:

```
...
Directions
----------
Using the long-tail method, cast on 24 stitches.

Row 1: Knit
Row 2: Purl
```

```
Repeat rows 1-2 for 4 in.

Row 25: Bind off all stitches
```

Here's another thought. A pattern designer may want to define more than one instruction group in the pattern. Think about knitting a sock, for instance. There are separate instructions for knitting the cuff, leg, heel flap, heel turn, gusset, foot, and toe. Being able to add headers to each instruction group would be very useful.

You can add instruction groups in the directions and label them.

```
pattern 'en' {
...
    InstructionGroup [label:'Cast On'] {
      CastOn 24 'long-tail'
    }
    InstructionGroup [label:'Make the Square'] {
      Instruction 'stockinette-st' {
        Row: k to end
        Row: p to end
      }
      Repeat 'stockinette-st' until measures 4 in
    }
    InstructionGroup [label:'Finish It Off!'] {
      Row: bindOff all sts
    }
...
}
```

This would look like:

```
Directions
----------
Cast On
-------
Using the long-tail method, cast on 24 stitches.

Make The Square
---------------
Row 1: Knit
Row 2: Purl
Repeat rows 1-2 for 4 in.

Finish It Off!
--------------
Row 25: bind off all stitches
```

Another quirk is that you probably would not see the bind off row expressed as Row 25. To reset the row count for this instruction group, add the keyword `resetRowCount` to the definition:

```
Pattern 'en' {
...
    InstructionGroup [label:'Finish It Off!'] resetRowCount {
      Row: BindOff all sts
    }
...
}
```

This will produce:

```
Finish It Off!
--------------
Row 1: Bind off all stitches
```

Alternatively, you may not want the row numbered at all, in which case you can do this:

```
Pattern 'en' {
...
    InstructionGroup [label:'Finish It Off!'] {
      NextRow: BindOff all sts
    }
...
}
```

and you will get:

```
...
Finish It Off!
--------------
Next Row: Bind off all stitches
```

Chapter 4. The XML Pattern

TODO

Chapter 5. Internationalization

TODO

Part III. Customizing and Extending KnitML

Table of Contents

Chapter 6. Customizing the Pattern Renderer

Overview

The `operations` resource bundle used to render patterns is embedded in the `knitml-pattern-renderer.jar` file. For your reference, the master resource bundle file has been copied to `KNITML_HOME/conf/local-operations.properties`. If you open it, you will see a list of name / value pairs, each separated by an equals sign and delineated by line. Each entry constitutes a phrase in a knitting pattern, with parameters supplied to it by the renderer and referenced by `{0}` (for the first parameter), `{1}` (for the second parameter), etc. You will base any customizations you make to the renderer on this file.

Personalizing Patterns

As knitters are a diverse group of individuals, knitters often use different notations and conventions to express the same technique. Sometimes terms vary regionally differ, and other times the techniques vary slightly but produce the same result.

Knitting a stitch so that it is twisted is a good example of a technique which can be described using different terms. Most knitters know this technique by the term "knitting through the back loop." This term, however, assumes that the mount of the stitch being worked is not reversed. While that generally is true if you are a western knitter, it is not necessarily true if you are a combination knitter. In combination knitting, if the same stitch in the previous row had been purled, the mount of that stitch would be reversed. If that is true, the back loop would *not* be the loop to knit to create a twisted stitch; it would be the front loop. It is more accurate to describe the loop to knit as the "trailing" loop instead of the "back" loop[1].

Another example is the yarn-over increase. Most commonly this is known by the abbreviation `yo` but can also be represented by `yfon` (yarn forward and over needle), `yfrn` (yarn forward and 'round needle), `yon` (yarn over needle), and `yrn` (yarn 'round needle).

You can instruct the renderer to use your favorite terms for knitting techniques instead of the defaults. Open the `custom-operations.properties` file in the `KNITML_HOME/conf` directory and copy over the entries you want to customize from `local-operations.properties`. A few samples are included in the comments (including the yarn-over customizations described above). To use one of them, simply remove the beginning # sign (which designates a comment) from the line you want to use.

Let's suppose a pattern I've rendered looks like this:

```
Row 1: k12, yo, k1
```

To make the yarn-over increase look like `yrn` (yarn 'round needle), I would open up `custom-operations.properties` and edit this line:

```
#operation.increase.yo=yrn
```

and make it look like this:

```
operation.increase.yo=yrn
```

Now my pattern would render like this:

[1] In fact, the associated entries in the `operations.properties` file that render a twisted knit stitch are tagged "through-trailing-loop" and are described as such in the XML form. This is because it is more accurate, not because it is a more common term.

```
Row 1: k12, yrn, k1
```

To change the way knits render, I could add this line:

```
operation.knit.$$$=k {0} st;k {0} sts
```

and it would look like this[2]:

```
Row 1: k 12 sts, yrn, k 1 st
```

There are several more examples of personalizations in the `custom-operations.properties` file.

Adding Support for a New Language

Configuring your Environment

To add a new language to the renderer, copy `KNITML_HOME/conf/local-operations.properties` to a file named `operations.properties` in the same directory. Next, tell the pattern renderer to use the local `operations` resource bundle rather than the one that's bundled with the pattern renderer. Do this by opening the `pattern-renderer-config.properties` file in the `KNITML_HOME/conf` directory and removing the # sign in front of the `messageSource.basename` property.

Next, open the `operations.properties` file you just created and translate the values (to the right of the equal signs) to the target language. The name of each entry aims to be fairly intuitive. Ine the case of more complex entries, a comment (a line starting with #) appears above the entry.

Handling Plurals

In short, every language has a set of plural forms. Many western languages use one form for the number 1, and a second form for everything else. This is by no means universal. Chinese, for instance, has only one plural form, while Scottish Gaelic has three and Irish has five! While there is no standard for dealing with this issue, Mozilla has developed a very practical approach to solving the problem, therefore we will adopt the same strategy here. Consult Localization and Plurals [https://developer.mozilla.org/en/Localization_and_Plurals] to determine your language's plural rule number.[3]. Once you have determined the plural rule number to use, set the number as the value of the `knitml.pattern-renderer.plural-rule` property at the top of the file.

Each entry in the `operations` file that ends with `.$$$` indicates that the value should be pluralized. Each plural form is specified between semi-colons. To write your language's plural forms, you can either write the phrase in a way which does not require a specialized form for the instance of the number, or you can write it in a way which requires explicit pluralized forms. In the case of the former, you only need to specify one value and that form will be used regardless of the plural form of the number. In the case of the latter, the first form occurs before the first semi-colon, the second form immediately after the first colon, the third form immediately after the second colon, etc.

For example, to localize the `operation.stitch.word` property, in English I could set the value either to `stitches: {0}` (no specific plural form) or `{0} stitch;{0} stitches` (plural form specific to the number).

It gets a bit trickier when there are two plural forms in the same phrase. Take the `operation.cross-stitches.front` property, for instance. The phrase looks something like "cross next X stitches over Y stitches." Because the pluralization for each number can vary, there are four different plural forms of this phrase in English (nine in Scottish Gaelic, twenty-five in Irish).

[2] Plurals are covered in the next section of this chapter.
[3] Most Western languages use plural rule #1. French uses either rule #1 or rule #2, depending on the region.

Since each number specifies an individual plural form, the entry in the property file is broken into two property names: `operation.cross-stitches.front.$$0` (tied to the pluralization of the first parameter) and `operation.cross-stitches.front.$$1` (tied to the pluralization of the second parameter). In the property file, it looks like this:

```
...front.$$0=cross next stitch;cross next {0} stitches
...front.$$1= over {1} stitch; over {1} stitches
```

The renderer concatenates both values together, using the `$$0` entry, then the `$$1` entry. If you want to specify a different order of concatenation, you can do that with a `$$order` parameter as follows:

```
...front.$$0= cross next stitch; cross next {0} stitches
...front.$$1=Over {1} stitch,;Over {1} stitches,
...front.$$order=10
```

This specifies that the `$$1` entry renders first, then the `$$0`, so it will render something like `Over 1 stitch, cross next 2 stitches`.

A `$$order` value of `01` in this case is the same as the default behavior (which is what happens if you do not provide the property).

Chapter 7. Integrating with APIs

This chapter will discuss writing your own software to integrate with KnitML's public APIs. This chapter will primarily be intended for programmers wishing to integrate their own knitting application with KnitML interfaces. A separate programmer's guide may be written instead.

Appendix A. The Specification

This section will detail the KnitML specification. For now, behavior is described fairly accurately in the XML Schema [schema/pattern-0.4.xsd.html].

Appendix B. Knitting Expression Language Reference

Table B.1. Defined Expression Language Functions

Function	Parameters	Description	Example	XML Equivalent
applyNextRow	ref as IDREF	Use the next row for the given block instruction	`applyNextRow 'my'`	`<apply-next-row instruction-ref="my"/>`
arrangeStitches	needleSpec as MapEntry... [39]	Arrange the stitches on the needles as specified, starting with the working stitch.	`arrangeStitches 'n1':10 'n2':12`	`<arrange-stitches-on-needles> <needle ref="n1">10</needle> <needle ref="n2">12</needle> </arrange-stitches-on-needles>`
ballWeight	number as Decimal, weight as WeightUnit	The weight of a single ball of yarn. The colon is optional.	`ballWeight: 50 g`	`<ball-weight unit="g">50 </ball-weight>`
bindOff	*(allStitches|all)* or number as Integer, stitches as Stitch (optional), *with* (optional), yarnRef as IDREF (optional), wise as Wise (optional)	Bind off either all of the stitches in the row / on the needle, or the specified number. Use the specified yarn and wise, if provided.	`bindOff 10 sts with 'y1' purlwise`	`<bind-off yarn-ref="y1" type="purlwise">10 </bind-off>`
			`bindOff all sts knitwise`	`<bind-off-all type="knitwise"/>`
bo / BO	(same as bindOff)	alias for bindOff	`bo 20`	`<bind-off> 20</bind-off>`
castOn	number as Integer, style as String, *countAsRow*, stitches as Stitch xmlAttributes as Map (all parameters optional in any order)	Cast the number of stitches onto the current needle. If *countAsRow* is specified, the cast on will be treated as a row (e.g. if it's the first row in a flat pattern, the next row in the pattern will be the wrong side).	`castOn 35 sts`	`<cast-on>35</cast-on>`
			`castOn 35 countAsRow`	`<cast-on count-as-row="false"> 35</cast-on>`
co / CO	(same as castOn)	alias for castOn	`co 35`	`<cast-on>35</cast-on>`
color	name as String, xmlAttributes as Map	The color of the yarn.	`color 'yellow' [number:127]`	`<color name="yellow" number="127"/>`

Function	Parameters	Description	Example	XML Equivalent
	(all parameters optional)			
copyrightInfo	(none)	Copyright information about this pattern. Not currently defined.	`copyrightInfo {...}`	`<copyright-info> ... </copyright-info>`
cross	first as Integer, crossSpec as Cross, next as Integer	Cross number of stitches specified by 'first' with number of stitches specified by 'next' using the crossSpec	`cross 2 inFrontOf 2`	`<cross-stitches first="2" next"2" type="front" />`
			`cross 2 behind 2`	`<cross-stitches first="2" next"2" type="back" />`
crossStitches	(same as cross)	alias for cross	`crossStitches 2 behind 2`	`<cross-stitches first="2" next"2" type="back" />`
declareFlatKnitting	*with*, side as Side, *next* (all parameters optional except side)	Start knitting flat. The next row facing is specified by the side parameter.	`declareFlatKnitting with rightSide next`	`<declare-flat-knitting next-row-side="right"/>`
declareRoundKnitting	(none)	Start knitting in the round.	`declareRoundKnitting`	`<declare-round-knitting/>`
designateEndOfRow	(none)	Signal that the last stitch worked is now the last stitch in the row.	`designateEndOfRow`	`<designate-end-of-row/>`
directions	(none)	The pattern directions	`directions {...}`	`<directions>...</directions>`
directives	(none)	The pattern directives	`directives {...}`	`<directives>...</directives>`
firstName	name as String	The pattern designer's first name. Falls within the `author` element. The colon is optional.	`firstName: 'Jonathan'`	`<first-name>Jonathan</first-name>`
forEachRowInInstruction	ref as IDREF	Apply the contained operations for every row in a given block instruction	`forEachRowInInstruction 'my' {...}`	`<for-each-row-in-instruction ref="my"> ... </for-each-row-in-instruction>`
fromHolder	ref as IDREF	Work the enclosed stitches from the specified stitch holder	`fromHolder 'sh1'{ p2, k to end }`	`<from-stitch-holder ref="sh1"> <purl>2</purl> <repeat until="end"> <knit/>`

Function	Parameters	Description	Example	XML Equivalent
				`</repeat>` `</from-stitch-holder>`
gauge	type as String (optional)	A gauge element	`gauge 'stockinette' {...}`	`<gauge type="stockinette"> ... </gauge>`
generalInformation	languageCode as String (optional)	The general information section written in the specified ISO language code	`generalInformation 'en' {...}`	`<general-information xml:lang="en"> ... </general-information>`

graftTogether	needles as IDREF... [39]	Graft together all of the needles specified by the parameters	`graftTogether 'n1' 'n2'`	`<graft-together>`
				`<needle ref="n1"/>`
				`<needle ref="n2"/>`
				`</graft-together>`

inform	stitchNumber as Integer, stitches as Stitch, stateDescription as StateDescription (optional)	Renders the number of stitches on the needles. Functionality may be expanded in the future.	`inform 20 sts leftOnNeedles`	`<information>`
				`<number-of-stitches number="20" inform="true"/>`
				`</information>`

			`inform 20`	`<information>`
				`<number-of-stitches number="20" inform="true"/>`
				`</information>`

Function	Parameters	Description	Example	XML Equivalent
informationalMessage	messageKey as String	Provide textual information about the related part of the pattern	`informationalMessage 'message.how-to-do-short-rows'`	`<information><message message-key = "message.how-to-do-short-rows"> ... </ message> </information>`
inlineInstruction	id as ID, *withKey*, *withLabel*, label as String, xmlAttributes as Map (all parameters except id optional)	Define an inline instruction	`inlineInstruction 'my' withKey withLabel 'Just do it' {...}`	`<inline-instruction id="my" message-key="inline-instruction.my" label="Just do it"> ... </inline-instruction>`
inlineInstructionRef	ref as IDREF	A reference to an inline instruction	`inlineInstructionRef 'my'`	`<inline-instruction-ref ref="my"/>`
instruction	id as ID, *withKey*, *withLabel*, label as String, shape as Shape, xmlAttributes as Map (all parameters except id optional)	Define a block instruction	`instruction 'my' withKey withLabel 'Just do it' round {...}`	`<instruction id="my" message-key="instruction.my" label="Just do it" shape="round"> ... </ instruction>`
instructionDefinitions	(none)	Global instruction definitions defined in the header. Can be block instructions, inline instructions, or merged instructions. Will appear in the legend before the directions.	`instructionDefinitions {...}`	`<instruction-definitions> ... </instruction-definitions>`
instructionGroup	id as ID, *withKey*, *withLabel*, label as String, xmlAttributes as Map (all parameters except id optional)	Define a group of block operations	`instructionGroup 'start' withKey withLabel 'Start Pattern' {...}`	`<instruction-group id="my" message-key="instruction-group.start" label="Start Pattern"> ... </instruction-group>`

Function	Parameters	Description	Example	XML Equivalent
instructionRef	ref as IDREF	A reference to a block instruction	`instructionRef 'my'`	`<instruction-ref ref="my"/>`
joinInRound	(none)	Join active stitches in the round.	`joinInRound`	`<join-in-round/>`
k	number as Integer, yarnRef as IDREF, loopToWork as LoopToWork, repeatSpec as Repeat, holderSpec as Holder (all parameters optional)	Knit specified number of stitches	`k10` `k 10`	`<knit>10</knit>`

			`k to end`
			`k end`
				`<repeat until="end">`
				`  <knit/>`
				`</repeat>`

			`k 3 'y1'`
			`k3 'y1'`
			`k 3 with 'y1'`
				`<knit yarn-ref="y1">3</knit>`

			`k to 2 sts before end`
			`k to 2 before end`
			`k 2 beforeEnd`
				`<repeat until="before-end"`
				`  value="2">`
				`  <knit/>`
				`</repeat>`

			`k 3 tbl` `k3 tbl`	`<knit loop-to-work="trailing">3</knit>`

Function	Parameters	Description	Example	XML Equivalent
k2tog	(none)	Knit next two stitches together	`k2tog`	`<decrease type="k2tog"/>`
kfb	(none)	Knit to the front and back of the next stitch.	`kfb`	`<increase type="kfb"/>`
knit	(same as k)	alias for k	`knit 10`	`<knit>10</knit>`
labelNeedle	id as ID, with as With, newLabel as String	labels a new needle with the newLabel parameter as either a message key (if keyword is 'withKey') or a literal value (if keyword is 'withLabel')	`labelNeedle 'n1' withKey 'needle.new-label'`	`<label-needle ref="n1" message-key = "needle.new-label"/>`
			`labelNeedle 'n1' withLabel 'Instep Needle'`	`<label-needle ref="n1" label = "Instep Needle"/>`
lastName	name as String	The pattern designer's last name. Falls within the `author` element. The colon is optional.	`lastName: 'Whitall'`	`<last-name>Whitall</last-name>`
length	number as Decimal, weight as LengthUnit	The length of a needle. Usually only given for circular needles. The colon is optional.	`length: 32 in`	`<length unit="in">32 </length>`
M1	(none)	Make 1. This does not specify a technique for increasing.	`M1`	`<increase/>`
m1a	(none)	Make 1 away	`m1a`	`<increase type="m1a"/>`
mergedInstruction	id as ID, *withKey*, *withLabel*, label as String, xmlAttributes as Map (all parameters except id optional)	Merge two existing instructions together (see Nutkin2 for a better example)	`mergedInstruction 'my' withKey withLabel 'Just do it' ['merge-point':'row', type:'physical'] {...}`	`<merged-instruction id="my" message-key="merged-instruction.my" label="Just do it" merge-point="row" type="physical"> ... </ merged-instruction>`
messageSource	source as IDREF	Use the specified ID as a localized message source for the pattern	`messageSource 'msgs'`	`<message-sources> <message-source>msgs </message-source>`

Function	Parameters	Description	Example	XML Equivalent
				`</message-sources>`

messageSources	sources as IDREF... [39]	Use the specified IDs as internationalized message sources for the pattern.	`messageSources 'header-msgs' 'body-msgs'`	`<message-sources>`
				`  <message-source>header-msgs`
				`  </message-source>`
				`  <message-source>body-msgs`
				`  </message-source>`
				`</message-sources>`

needle	id as ID, *withKey*, *withLabel*, label as String, xmlAttributes as Map (all parameters except id and typeref optional)	A needle used in the pattern.	`needle 'needle1' withKey withLabel 'Needle 1' [typeref:'size1'] {...}`	`<needle id="needle1" message-key="needle.needle1" label="Needle 1" typeref="size1"> ... </needle>`
needleType	id as ID, needleStyle as NeedleType (optional), xmlAttributes as Map (optional)	Define a type of needle for use in the pattern.	`needleType 'size1' circ [brand:'Addi Turbo']`	`<needle-type id="size1" type="circular" brand="Addi Turbo"> ... </needle-type>`
nextRound	(same as row)	call row() with supplied parameters, adding ['assign-row-number':false] and [type:'round'] to the xmlAttributes parameter	`nextRound {...}`	`<row assign-row-number="false" type="round">` `...</row>`
nextRow	(same as row)	call row() with supplied parameters, adding ['assign-row-number':false] to the xmlAttributes parameter	`nextRow {...}`	`<row assign-row-number="false">` `...</row>`

p	number as Integer, yarnRef as IDREF, repeatSpec as Repeat
	
	(all parameters optional)
		Works the same as 'k' except that the XML element name is `purl`	`p to end`	`<repeat until="end">`
				`  <purl/>`
				`</repeat>`

Function	Parameters	Description	Example	XML Equivalent
p2tog	(none)	Purl next two stitches together	`p2tog`	`<decrease type="p2tog"/>`
pattern	languageCode as String (optional)	The pattern written in the specified ISO language code	`pattern 'en' {...}`	`<pattern xml:lang="en"> ... </pattern>`
pickUp	number as Integer, stitches as Stitch, yarnRef as IDREF, wise as Wise (all parameters except number optional)	Pick up the specified number of stitches. If wise is not specified, defaults to knitwise. Renders as `inline-pickup-stitches` if within a row.	`pickUp 10 sts 'y1' purlwise`	`<pick-up-stitches type="purlwise" yarn-ref="y1">10</pick-up-stitches>`
			`pickUp 20`	`<pick-up-stitches>20</pick-up-stitches>`
placeMarker	(same as pm)	alias for pm	`placeMarker`	`<place-marker/>`
pm	(none)	Place a marker at the working place in the pattern.	`pm`	`<place-marker/>`
purl	(same as p)	alias for p	`purl 10`	`<purl>10</purl>`
ref	ref as IDREF	alias for inlineInstructionRef	`ref 'my'`	`<inline-instruction-ref ref="my"/>`
removeMarker	(none)	Remove the marker at the working place in the pattern.	`removeMarker`	`<remove-marker/>`
repeat	repeatSpec as Repeat, xmlAttributes as Map	Repeat the enclosed inline instructions as many times as specified	`repeat 3 times {...}`	`<repeat until="times" value="3"> ... </repeat>`
repeatInstruction / repeat	ref as IDREF, repeatSpec as RepeatInst, xmlAttributes as Map	Repeat the referenced instruction as many times as specified	`repeat 'inst1' until 20 sts remain`	`<repeat-instruction ref="inst1"><until-stitches-remain>20</until-stitches-remain></repeat-instruction>`
			`repeat 'inst1' until measures 4 in`	`<repeat-instruction ref="inst1"><until-measures unit="in">4</until-measures>`

Function	Parameters	Description	Example	XML Equivalent
				</repeat-instruction>
			repeat 'inst1' 8 additionalTimes	<repeat-instruction ref="inst1"> <additional-times>8 </additional-times> </repeat-instruction>
round	(same as row)	call row() with supplied parameters, adding [type:'round'] to the xmlAttributes parameter	round {...}	<row type="round">...</row>
row	yarnRef as IDREF, number as Integer, numbers as List<Integer>, *informSide*, *doNotAssignNumber*, xmlAttributes as Map (all parameters optional)	A row in the pattern	row 'y1' 1 {...}	<row yarn-ref="y1" number="1">...</row>
			row [1,3,5] {...}	<row number="1 3 5">...</row>
			row doNotAssignNumber {...}	<row assign-row-number="false">...</row>
			row informSide {...}	<row inform-side="true">...</row>
			row [type:'flat', resetRowCount:true] {...}	<row type="flat" reset-row-count="true">...</row>
rowGauge	number as Decimal, gauge as RowGaugeUnit	The number of rows (vertically) per unit of measurement. The colon is optional.	rowGauge: 3.5 rowsPerInch	<row-gauge unit="row/in">3.5 </row-gauge>
section	*resetRowCount* (optional)	A logical grouping of operations in a pattern. Can be interpreted as a paragraph. The resetRowCount parameter	section resetRowCount {...}	<section reset-row-count="true"> ... </section>

Function	Parameters	Description	Example	XML Equivalent
		sets the next row back to row 1.		
shortRow	(same as row)	call row() with supplied parameters, adding `[short:true]` to the xmlAttributes parameter	`shortRow 1 {...}`	`<row short="true" number="1">...</row>`
size	number as Decimal or String, size as NeedleSize	The size (i.e. thickness) of the needle.	`size 3.0 mm`	`<size unit="mm">3.0 </size>`
			`size '00' US`	`<size unit="US">00 </size>`
sl	*next*, number as Integer, stitches as Stitch, wise as Wise, *reverse* | *inReverse*, position as YarnPosition, xmlAttributes as Map (all parameters optional)	Slip the specified number of stitches	`sl 5`	`<slip>5</slip>`
			`sl 5 sts knitwise`	`<slip type="knitwise">5 </slip>`
			`sl next 5 sts to holder 'h1'`	`<slip-to-stitch-holder ref="h1"> </slip-to-stitch-holder>`
			`sl 5 inReverse`	`<slip direction="reverse">5 </slip>`
			`sl 5 wyib`	`<slip yarn-position="back">5 </slip>`
ssk	(none)	Slip-slip-knit	`ssk`	`<decrease type="ssk"/>`

state	stitchNumber as Integer, stitches as Stitch, stateDescription as StateDescription... (all parameters optional in any order)	State the number of stitches left in the row, or state the number of stitches which have yet to be worked for this row. If a number is specified, the validator asserts that this	`state 10 sts rem`
			`state 10 sts remain`
			`state 10 sts remaining`
			`state 10 sts left`
			`state 10 sts inRow`
			`state 10 sts onNeedles`
			`state 10 sts left onNeedles`
				`<information>`
				`  <number-of-stitches`
				`  number="10"/>`
				`</information>`

Function	Parameters	Description	Example	XML Equivalent
		number matches reality. Can be used at the end of a row definition, in which case the element name will be `<followup-information>` instead of `<information>`.	`state 10 sts`	

			`state sts rem`
			`state rem sts`
			`state remaining sts`
			`state sts`
				`<information>`
				`  <number-of-stitches/>`
				`</information>`

			`state 10 unworked sts`
			`state 10 sts unworked`
			`state 10 unworked`
				`<information>`
				`  <number-of-unworked-stitches number="10"/>`
				`</information>`

			`state unworked sts`
			`state sts unworked`
			`state unworked`
				`<information>`
				`  <number-of-unworked-stitches/>`
				`</information>`

Function	Parameters	Description	Example	XML Equivalent
stitchGauge	number as Decimal, gauge as StitchGaugeUnit	The number of stitches (horizontally) per unit of measurement. The colon is optional.	`stitchGauge: 3.5 stitchesPerInch`	`<stitch-gauge unit="st/in">3.5 </stitch-gauge>`
stitchHolder	id as ID, *withKey*, *withLabel*, label as String (all parameters except id optional)	A stitch holder used in the pattern.	`stitchHolder 'sh1'`	`<stitch-holder id="sh1"/>`
			`stitchHolder 'sh1' withKey withLabel "Stitch Holder 1"`	`<stitch-holder id="sh1" message-key="stitch-holder.sh1" label="Stitch Holder 1"/>`
thickness	number as Decimal, thickness as ThicknessUnit	The thickness of the yarn (wraps per unit of measurement). The colon is optional.	`thickness: 16 wrapsPerInch`	`<thickness unit="wrap/in"> 16</thickness>`
totalLength	number as Decimal, weight as LengthUnit	The length of yarn required to complete the project. The colon is optional.	`totalLength: 500 yd`	`<total-length unit="g">50 </total-length>`
totalWeight	number as Decimal, weight as WeightUnit	The weight of yarn required to complete the project. The colon is optional.	`totalWeight: 50 g`	`<total-weight unit="g">50 </total-weight>`
useNeedle	*quietly* (optional), needle as IDREF	Use the specified needle for knitting. "Quietly" indicates that this is a directive for the computer, not something that should be rendererd in a pattern.	`useNeedle 'n1' quietly`	`<use-needles silent="true"> <needle ref="n1"/> </use-needles>`
useNeedles	*quietly* (optional), needles as IDREF... [39]	Use the specified needles for knitting.	`useNeedles 'n1' 'n2'`	`<use-needles silent="true"> <needle ref="n1"/> <needle ref="n2"/> </use-needles>`
usingNeedle	ref as IDREF	Using the specified needle, perform child operations	`usingNeedle 'needle1' {...}`	`<using-needle ref="needle1"> ... </using-needle>`

Function	Parameters	Description	Example	XML Equivalent
yarn	id as ID, *withKey*, *withLabel*, label as String, xmlAttributes as Map (all parameters except id optional)	A yarn used in the pattern. The xmlAttributes parameter must have a typeref attribute referring to a previously declared yarnType.	`yarn 'y1' withKey withLabel 'Main Color' [symbol:'MC', typeref:'wool'] {...}`	`<yarn id="y1" message-key="yarn.MC" label="Main Color" symbol="MC" typeref="wool"> ... </yarn>`
yarnType	id as ID, xmlAttributes as Map	Define a type of yarn for use in the pattern.	`yarnType 'wool' [brand:'Lorna \'s Laces', weight:'fingering'] {...}`	`<yarn-type id="wool" brand="Lorna's Laces" weight="fingering"/>...</ yarn-type>`
yo	(none)	Yarn over	`yo`	`<increase type="yo"/>`

The ... notation indicates that there may be any number of individual parameters with this name and type.

Whether or not the function is defined in Table B.1, “Defined Expression Language Functions”, you can use the following syntax to define any XML element:

`elementName` `[`*`attr1`*`:'`*`attr1value`*`',`*`attr2`*`:'`*`attr2value`*`'...] {` *`body`* `}`

When an element or attribute name has a hyphen in it (such as `instruction-group`), it must be in single quotes for the processor to interpret it correctly.

The following table shows a few examples of custom functions:

Table B.2. Custom Function Examples

KnittingEL	XML

`author {`
`  'first-name' Jonathan`
`  'last-name' Whitall`
`}`
	`<author>`
	`  <first-name>Jonathan</first-name>`
	`  <last-name>Whitall</last-name>`
	`</author>`

`instruction [id:'my','message-key':'instruction.my'] {`
`   k1, p1`
`}`
	`<instruction id="my"`
	`  message-key="instruction.my">`
	`  <knit>1</knit>`
	`  <purl>1</purl>`

KnittingEL	XML
	`</instruction>`
`increase [type:'moss']`	`<increase type="moss"/>`
`slip5`	`<slip>5</slip>`
`'slip5'` (not valid KnitML)	`<slip5/>`

`tree [age:'150',size:'big'] {`
`  name: 'Old Faithful'`
`}`
 	
(not valid KnitML)
	`<tree age="150" size="big">`
	`  <name>Old Faithful</name>`
	`</tree>`

Use the pre-defined functions whenever possible, but there may be times when you need maximum flexibility to express the resulting XML.

Table B.3. Expression Language Data Types

Type	Form	Example
Cross	`('inFrontOf' | 'behind')`	`inFrontOf`
Decimal	Same as XML Schema's decimal data type	`1.5`
Holder	`('to' | 'from') 'holder' holderId`	`from holder 'stitch-holder-1'`
		`to holder 'stitch-holder-1'`
ID	Same as XML Schema's ID data type	`'make-cuff'`
IDREF	Same as XML Schema's IDREF data type	`'make-cuff'`
Integer	Same as XML Schema's short data type	`3`
LengthUnit	`('in' | 'inch' | 'ins' | 'inches' | 'cm' | 'centimeters' | 'mm' | 'millimeters' | 'm' | 'meters' | 'yd' | 'yards')`	`inch`
LoopToWork	`('tbl')`	`tbl`
Map	Grouping of name / value pairs; always translates to a list of attributes for the target element. Each name is taken	`[id:'needle1', typeref:'needle-type1', 'message-key':'needle.needle1']`

Type	Form	Example
	literally and need not be in quotes (unless there is a hyphen in the name).	
MapEntry	A name / value pair. The name is taken literally and need not be in quotes (unless there is a hyphen in the name).	`needle1:36`
NeedleSize	`('US' | 'mm')`	`US`
NeedleType	`('circ' | 'circular' | 'straight' | 'dpn')`	`circ`
Repeat	`('to'?, Integer?, 'before'?, Until)` in any order	`to 2 before end`
		`to end`
		`8 times`
RepeatInst	`('until'?, UntilInst, (Decimal | Integer)?, Stitch?, LengthUnit?)` in any order	`until measures 4.5 in`
		`until 20 stitches remain`
		`until 1 st remains`
		`8 additionalTimes`
RowGaugeUnit	`('rowsPerInch' | 'rowsPerCentimeter')`	`rowsPerInch`
Side	`('right' | 'wrong')`	`wrong`
Shape	`('flat' | 'round')`	`round`
StateDescription	`('rem' | 'remain' | 'remaining' | 'inRow' | 'onNeedle' | 'onNeedles' | 'left' | 'unworked')`	`remaining`
Stitch	`('st' | 'sts' | 'stitch' | 'stitches')`	`sts`
StitchGaugeUnit	`('stitchesPerInch' | 'stitchesPerCentimeter')`	`stitchesPerCentimeter`
String	Same as XML Schema's string data type	`'hello'`
ThicknessUnit	`('wrapsPerInch' | 'wrapsPerCentimeter')`	`wrapsPerInch`
Until	`('end' | 'beforeEnd' | 'beforeGap' | 'marker' | 'beforeMarker' | 'times')`	`marker`

Type	Form	Example
UntilInst	('measures' | 'desiredLength' | 'remain' | 'remains' | 'additionalTimes')	desiredLength
WeightUnit	('g' | 'grams' | 'oz' | 'ounces')	grams
Wise	('knitwise' | 'purlwise')	purlwise
With	('withKey' | 'withLabel')	withKey
YarnPosition	('wyif' | 'wyib')	wyif